UQAC
UNIVERSITÉ DU QUÉBEC
À CHICOUTIMI

MÉMOIRE

PRÉSENTÉ À L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI COMME EXIGENCE

PARTIELLE DE LA MAÎTRISE EN INFORMATIQUE

PAR

EDMOND LA CHANCE, B.Sc

ALGORITHMES POUR LE PROBLÈME DE L'ARBRE COUVRANT MINIMAL

2014

Ce travail de recherche a été réalisé
à l'Université du Québec à Chicoutimi
dans le cadre du programme
de la Maîtrise en informatique

Pour l'obtention du grade : Maître ès sciences. M. Sc.

Résumé

Le problème de l'arbre couvrant minimal est un des plus vieux problèmes en théorie des graphes. La problématique se pose comme suit : étant donné un graphe avec un nombre de sommets et un nombre d'arêtes ayant des poids de valeurs dans l'ensemble des entiers relatifs, l'arbre couvrant minimal consiste à trouver l'ensemble des arêtes permettant de rejoindre tous les sommets sans former de cycle, et ce, avec un coût minimal. Ce problème trouve des applications pratiques variées : il est directement applicable pour l'optimisation et la conception de divers types de réseaux (électrique, internet, etc.). Son application la plus populaire est actuellement dans le domaine du forage de données. La raison étant que certains algorithmes construisent l'arbre minimal en faisant grossir des groupes d'éléments progressivement. Ces groupes, appelés également clusters, représentent ensuite des groupes d'éléments similaires, ce qui permet de faire du forage de donnée efficacement. Il suffit au préalable de transformer les données multidimensionnelles du problème de clustering en graphe.

Pour trouver l'arbre couvrant minimal, de nombreux algorithmes ont été proposés dans la littérature et ils sont pour la plupart assez performants. Cependant, avec l'invention récente de meilleures structures de données et de nouvelles techniques, les algorithmes n'ont cessé d'évoluer dans les vingt dernières années. Se pose donc la question : quel est l'algorithme le plus performant d'un point de vue

théorique et/ou pratique? Les algorithmes testés dans ce mémoire ont des complexités temporelles assez semblables ; il est donc difficile de prévoir *a priori* celui qui sera le plus performant. Nous avons pour cela choisi une série d'algorithmes résolvant ce problème, nous avons expliqué en détail leur fonctionnement et nous les avons programmés en utilisant plusieurs structures de données. Ces algorithmes sont ceux de Kruskal, Borůvka et Prim. L'algorithme de Prim a été implémenté avec de nombreuses files de priorité et les algorithmes de Kruskal et Borůvka utilisent les meilleurs algorithmes de gestion des ensembles disjoints. Nous avons ensuite fait des tests empiriques sur des graphes de différentes tailles et densités afin de pouvoir confirmer les avantages de leur complexité temporelle. Les résultats obtenus permettent de tirer de nombreuses conclusions sur les performances de chaque algorithme en fonction de la densité du graphe. L'algorithme de Borůvka sort notamment du lot, car il propose des performances excellentes pour toutes les densités de graphe.

Remerciements

Je remercie sincèrement mon directeur Djamal Rebaïne pour les nombreuses corrections et ses conseils sur la méthodologie des tests, la complexité algorithmique et pour tout le temps qu'il m'a consacré.

Je remercie également mon père Michäel pour son aide précieuse.

LISTE DES FIGURES

LISTE DES ALGORITHMES

LISTE DES TABLEAUX

INTRODUCTION GÉNÉRALE

De nombreuses recherches ont été faites sur l'optimisation des problèmes de graphes. Les graphes permettent de représenter des ensembles d'éléments qui sont connectés d'une certaine façon. Prenons le graphe d'un réseau routier, chaque sommet est un élément de l'ensemble des villes et chaque arête définit la relation, la connexion entre chaque ville. Le graphe est donc une structure mathématique de base qui permet de modéliser des problèmes de la vie réelle.

Le problème de l'arbre couvrant minimal (également appelé en anglais MST pour Minimum Spanning Tree) est un des plus anciens problèmes en théorie des graphes. Ce problème se résume à trouver un arbre qui connecte tous les nœuds d'un graphe à l'aide d'un ensemble d'arêtes dont le coût est minimal. Le problème a une histoire riche, car de nombreux algorithmes existent pour calculer l'arbre couvrant minimal d'un graphe. L'algorithme de Borůvka [Graham and Hell 1985] a été conçu en 1926 par un mathématicien du nom de Otakar Borůvka pour trouver un réseau électrique optimal pour la Moldavie, un pays situé en Europe Centrale. L'algorithme de Prim a été conçu en 1930 par le mathématicien Jarnik [Graham and Hell 1985] et est ensuite redécouvert indépendamment par Prim [Prim 1957]. L'algorithme de Kruskal [Kruskal 1956] a été quant à lui conçu en 1956. Les algorithmes déterminant l'arbre couvrant minimal sont intéressants, car ils ont souvent une application secondaire.

À titre d'exemple, mentionnons l'application du forage de données qui consiste à extraire de l'information pertinente depuis des données brutes de façon automatique. Dans ce domaine, il y a trois groupes d'algorithmes: les algorithmes de classification, les algorithmes de clustering et les algorithmes d'association. Un algorithme de clustering va grouper des éléments jugés similaires à l'aide d'une fonction de distance. L'algorithme de Kruskal et l'algorithme de Borůvka sont intéressants pour faire du clustering car ils fonctionnent en construisant des groupes distincts d'objets, ce qui est très utile pour cette application. La fonction de distance permet de créer des arêtes entre chaque élément et ensuite l'application de l'algorithme de MST, étape par étape, permet de trouver les clusters.

Dans ce mémoire, nous commençons par expliquer le fonctionnement des trois grands algorithmes classiques, à savoir ceux de Borůvka, Kruskal et Prim. Ensuite, nous présentons et discutons leurs performances à travers une étude empirique. Nous avons utilisé la méthodologie de Berenbaum [Berenbaum 1998]. Cette méthodologie teste tous les algorithmes avec des graphes générés aléatoirement ayant des densités variables, ce qui permet d'avoir un portrait complet du fonctionnement de l'algorithme étudié.

L'organisation de ce mémoire, divisé en trois chapitres, est comme suit. Le Chapitre 1 présente le fonctionnement des algorithmes de Prim, Kruskal et Borůvka que nous avons utilisés dans la résolution de la problématique de l'arbre couvrant de poids minimal. Le Chapitre 2 présente les différentes files de priorité utilisées dans la littérature dont nous avons besoin pour l'implémentation des différents algorithmes

utilisés dans ce mémoire. Au Chapitre 3, nous discutons de l'étude expérimentale que nous avons menée pour comparer la performance des algorithmes cités précédemment. En guise de conclusion, nous résumons et présentons nos remarques et commentaires sur le travail que nous avons effectué.

CHAPITRE 1: ARBRE COUVRANT MINIMAL

1.1 *Introduction*

Nous allons définir le problème de l'arbre couvrant minimal et détailler les algorithmes existants pour sa résolution. Nous parlerons d'abord des algorithmes UNION-FIND car ces algorithmes permettent de travailler d'une manière efficace avec des ensembles disjoints. Les ensembles disjoints sont utilisés par les algorithmes de Borůvka et Kruskal. Ensuite, nous expliquerons les algorithmes classiques de Borůvka, Kruskal [Kruskal 1956] et Prim [Prim 1957]. Leur fonctionnement est simple et ce sont les algorithmes couramment utilisés car leur implémentation sur une machine est plus facile à mettre en œuvre. Nous allons également discuter des structures de données qui accompagnent ces algorithmes. Le tas de Fibonacci [Fredman and Tarjan 1987] est très intéressant pour une bonne performance avec l'algorithme de Prim. Nous détaillerons le fonctionnement de ces algorithmes à l'aide d'illustrations, d'explications et de preuves mathématiques. Notons que les notions relatives à la complexité algorithmique sont résumées à la fin de ce mémoire dans Annexe I.

1.2 *Graphes, arbres et arbres couvrants*

Un graphe G = (V, E) est une collection de n sommets et m arêtes. Une arête définit une relation entre deux sommets. Dans le cadre de notre problématique, les arêtes représentent la distance entre deux sommets.

Un arbre est un graphe acyclique et connexe. Berenbaum [Berenbaum 1998] mentionne que les trois propriétés les plus importantes d'un arbre sont :

1. Un arbre avec n nœuds a exactement n-1 arêtes.
2. Quand une arête est ajoutée à un arbre, un cycle est créé dans le graphe. Ce qui contredit la propriété 1 (un arbre ne peut avoir n arêtes). Un graphe avec un cycle ne peut pas être un arbre.
3. Si un arbre perd une arête, il devient déconnecté. Un arbre qui perd une arête se transforme en deux arbres.

Si un graphe G contient n sommets et m arêtes, l'arbre couvrant correspondant contient n sommets et $n-1$ arêtes. Quand le graphe est au maximum de densité, $m = n(n-1)/2$, il y a un nombre important d'arêtes à enlever pour que tous les sommets soient connectés avec un nombre minimum d'arêtes pour générer un arbre couvrant de poids minimum.

L'arbre couvrant minimal est, comme son nom l'indique, un problème de minimisation. Il s'agit de sélectionner un arbre utilisant tous les sommets du graphe dont le poids total est minimum. La somme $P(T)$ des poids d'un arbre couvrant T est représentée par la somme des poids des arêtes. La fonction $w(u,v)$ retourne le poids de l'arête connectant les sommets u et v.

$$P(T) = \sum_{(u,v)\in T} w(u,v).$$

1.3 *Propriétés d'un arbre couvrant minimal*

Un arbre couvrant minimal respecte toujours certaines propriétés. Ces propriétés sont toujours vraies pour tous les MST. Connaître ces propriétés est très important,

car de nombreux algorithmes y font référence pour expliquer la logique derrière leur fonctionnement.

- **Propriété des multiples possibles**
  - Si certaines arêtes ont le même poids, il est alors possible qu'il y ait plusieurs MST équivalents en poids pour un graphe G.
- **Propriété de l'unicité**
  - Si toutes les arêtes ont des poids distincts, alors il n'y a qu'un seul MST.
- **Propriété du cycle**
  - S'il y a un cycle dans le graphe, alors l'arête la plus lourde de ce cycle ne fait pas partie du MST.
- **Propriété de la coupure**
  - La coupure d'un graphe connecté est un ensemble d'arêtes qui, une fois enlevées, séparent le graphe en deux composantes disjointes. La propriété de la coupure stipule que la plus petite arête de la coupure fait partie du MST.
- **Propriété de l'arête de poids minimum**
  - Si un sommet u possède une arête de poids minimum et unique, cette arête fait toujours partie du MST.

1.4 *Algorithmes d'UNION-FIND*

Les algorithmes d'UNION-FIND sont utilisés par les algorithmes classiques de MST (Kruskal, Borůvka). Ces algorithmes construisent le MST en ajoutant une arête et en détectant si cette arête crée un cycle ou non. Les cycles peuvent être détectés à l'aide des algorithmes d'UNION-FIND car ces derniers manipulent des structures

appelées ensembles disjoints. Les ensembles disjoints permettent dans le cadre du problème du MST de gérer des groupes de sommets et ensuite savoir si une arête crée un cycle en vérifiant si cette dernière lie un groupe avec lui-même ou alors lie un groupe avec un autre groupe. Les algorithmes de UNION-FIND reposent sur deux opérations essentielles : RECHERCHE(x) et FUSION(x,y). L'opération recherche permet d'obtenir le groupe d'un élément x. L'opération Fusion permet de fusionner deux groupes. La Figure 1 montre le fonctionnement d'une telle structure.

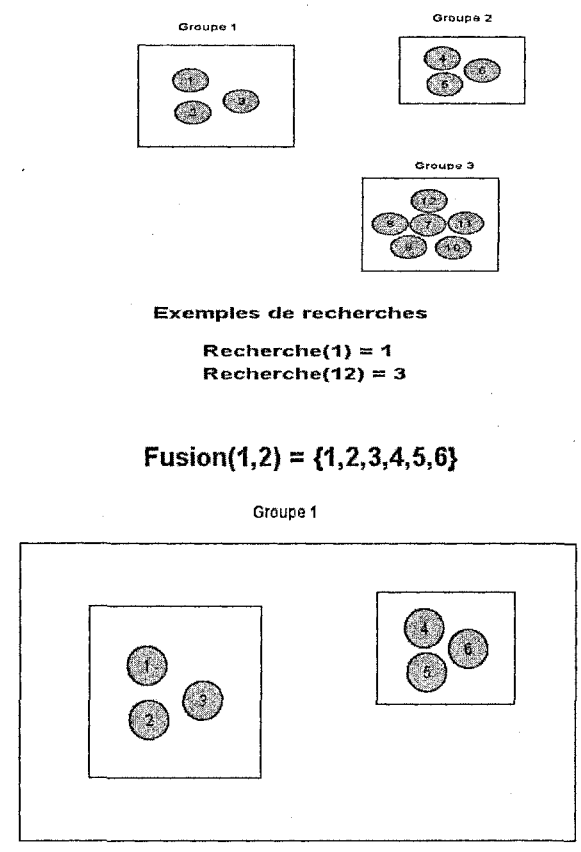


Figure 1 : Ensembles disjoints

Lors de la fusion de deux ensembles disjoints, plusieurs stratégies peuvent être utilisées. L'algorithme le plus performant d'UNION-FIND utilise une compression par chemin, ce qui donne une complexité amortie de $O(\alpha(m))$, la fonction Ackermann inverse. La fonction Ackermann inverse est une fonction qui croit beaucoup plus lentement qu'un logarithme. Cet algorithme est expliqué dans la Méthode 2. La méthode 1 propose une solution plus simple à implémenter.

1.4.1 **Méthode 1** : Tableau de correspondance entre un sommet et son groupe

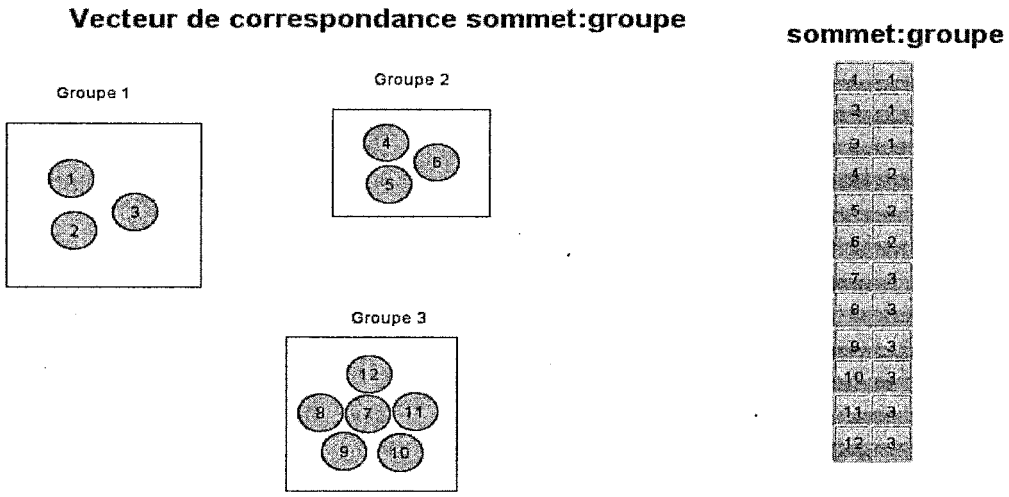


Figure 2 : Vecteur de correspondance

Cet algorithme utilise un vecteur de correspondance (Figure 2) entre un sommet et son groupe. Les indices du vecteur sont les sommets et la valeur du vecteur à un indice est le groupe correspondant. Lors de la fusion du groupe 1 et du groupe 2, tout le vecteur est traversé et les entrées sont changées si cela est nécessaire. La

complexité temporelle de cet algorithme est en $O(n)$. Dans l'Algorithme 1, l'ensemble a est absorbé dans l'ensemble b. C'est un choix arbitraire pour l'algorithme. Une stratégie d'absorption, c'est-à-dire la question de quel groupe survit et quel groupe meurt n'est pas nécessaire pour cet algorithme car la fusion prend toujours un temps $O(n)$ quoi qu'il arrive. Voici l'algorithme de fusion pour l'UNION-Find simple.

```
Fonction Fusion(a,b)
{
Répéter pour n sommets
    Soit i le sommet courant,
SI groupe[i] == a
            groupe[i] = b;
} //fin fonction
```

Algorithme 1 : UNION-FIND simple

1.4.2 Méthode 2 : Compression par chemin

L'algorithme d'UNION-FIND basé sur la compression de chemin est une version beaucoup plus rapide que la technique précédente. Cet algorithme a été conçu dans l'optique d'être plus rapide sur un grand nombre d'opérations. Au lieu d'avoir un simple vecteur de correspondance, cet algorithme construit un graphe des fusions de chaque groupe et réduit la taille de celui-ci régulièrement avec son algorithme de compression de chemin. Une heuristique de rang peut également être ajoutée pour que les fusions choisissent le groupe avec la plus grande population comme successeur. Cette structure de graphe à l'avantage de pouvoir être implémentée avec un simple vecteur et non une structure à pointeurs, ce qui assure une facilité d'implémentation et une grande vitesse à cet algorithme.

La compression de chemin se produit lors des requêtes de recherche sur la structure UNION-FIND. Lorsque le vrai groupe d'un élément est trouvé, tous les sous-groupes sont mis à jour pour pointer vers leur vrai groupe. Ce qui a comme effet d'aplatir tout le chemin car la liste chainée de groupes est remplacée par un lien direct vers le vrai groupe. Les requêtes de recherche subséquentes sont donc plus rapides.

Cet algorithme peut-être implémenté avec une fonction récursive de la façon suivante. La récursivité et la pile d'exécution sont utilisées pour voyager agilement à travers la structure UNION-FIND. Dans cette fonction, le paramètre x représente un groupe. Le vecteur groupes est un tableau d'une taille linéaire du nombre de groupes nécessaires.

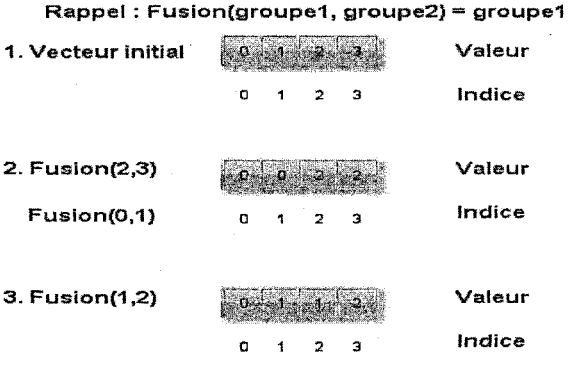
```
Fonction TrouverGroupeParent(x)
{
  SI groupes[x] == x
    return x ;
  SINON
    groupes[x] = TrouverGroupeParent(groupes[x]) ;
  return groupes[x] ;
} //fin de la fonction
```

Algorithme 2 : Compression par chemin

Tarjan [Tarjan 1975] a montré à l'aide d'une analyse en complexité amortie (voir Annexe I) qu'un algorithme de compression par chemin est de complexité temporelle en $O(\alpha(m))$, ce qui correspond à la fonction Ackermann inverse. La fonction Ackermann est une fonction qui croit très rapidement, son inverse croit très lentement. À toutes fins pratiques, la valeur de $\alpha(m)$ peut être vue comme une

constante de 5 car les valeurs de m utilisées ne seront jamais assez grandes en pratique pour dépasser 5.

Nous allons maintenant illustrer le processus de compression par chemin avec des schémas qui montrent comment ces algorithmes fonctionnent.



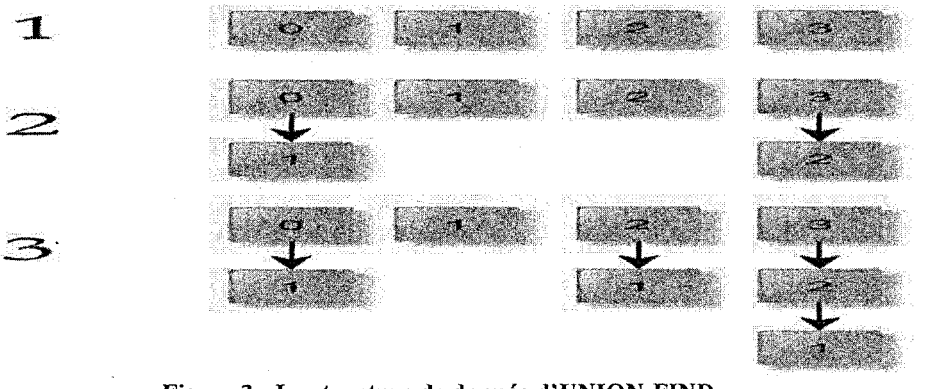


Figure 3 : La structure de donnée d'UNION-FIND, vue comme un vecteur ou un graphe

Dans la Figure 3, nous avons représenté de deux façons la structure des ensembles disjoints pour la compression par chemin : avec un vecteur et un graphe. Dans le vecteur, l'indice représente le numéro du groupe et la valeur du vecteur à cet

indice représente l'allégeance de ce groupe. Si le groupe d'indice 2 à comme valeur 2, cela veut dire qu'il n'a pas été assimilé dans aucun groupe. Ce groupe n'a pas de parent. Par contre, si la valeur du vecteur à l'indice 2 est de 1, cela veut dire que le groupe 2 a été assimilé dans le groupe 1.

L'étape 1 de cet exemple représente un graphe avec 4 ensembles disjoints. À l'étape 1, pour fusionner le groupe 3 avec le groupe 2 et le groupe 0 avec le groupe 1, il faut simplement remplacer la valeur du groupe qui est fusionné. La fusion de groupes se fait donc en temps constant avec cet algorithme. À l'étape 3, la même chose se produit, mais il y a maintenant 2 niveaux de profondeur. Regardons maintenant comment la compression de chemin fonctionne.

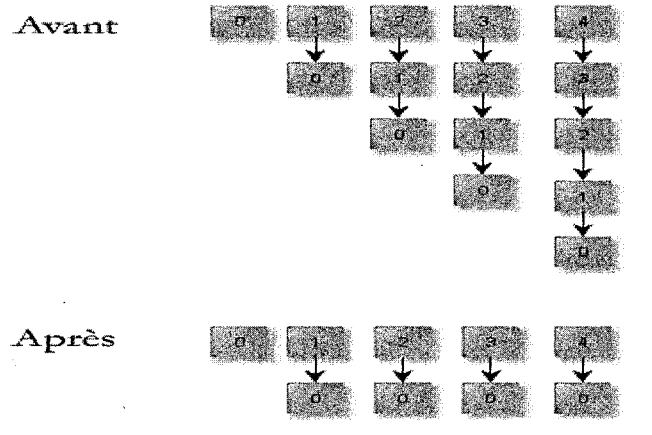


Figure 4 : Compression du chemin après une itération

À l'aide de la Figure 4, nous présentons un exemple montrant comment le chemin est compressé. En effet, si nous voulons trouver le parent du groupe 4, nous allons commencer par suivre la chaîne des parents jusqu'à ce que nous trouvions que

le vrai parent du groupe 4 qui est le groupe 0. Une fois cette information trouvée, quand nous déroulons la pile des parents intermédiaires, nous informons ceux-ci que leur parent final est le groupe 0. Le groupe 2 pointera donc directement sur le groupe 0 et ainsi de suite. Tout le chemin a été aplati et les prochaines requêtes de recherche sur la structure UNION-FIND seront répondues en temps $O(1)$.

1.5 *Algorithme de Borůvka*

L'algorithme de Borůvka est l'algorithme le plus ancien qui existe pour trouver l'arbre couvrant minimal (MST). Cet algorithme utilise la propriété de l'arête de poids minimum. Cette propriété nous donne l'information suivante : la plus petite arête qu'un sommet possède est dans le MST. Nous pouvons donc commencer avec le graphe complet, nous visitons chaque sommet et nous colorions la plus petite arête connectée à celui-ci.

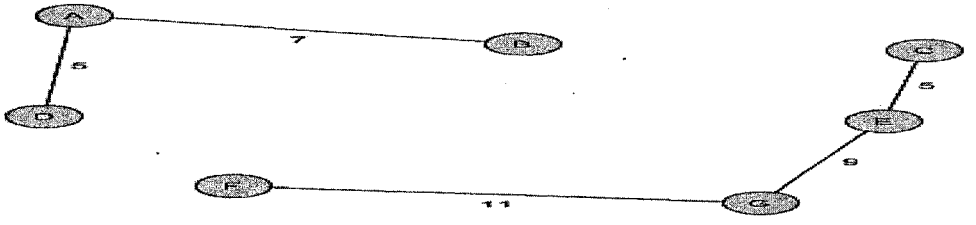


Figure 5 : Deux ensembles disjoints

Après avoir effectué cette première étape, il y a maintenant un bon nombre d'arêtes qui font partie du MST. Cependant, même si maintenant tous les sommets ont au moins une arête les atteignant, le MST n'est toujours pas trouvé. Cette étape a trouvé un bon nombre d'ensembles disjoints dans le graphe que nous pouvons voir

comme des petites îles isolées. La Figure 5 montre deux ensembles disjoints qui ont été créés après une étape de l'algorithme de Borůvka. Maintenant, si cette étape est réappliquée une deuxième fois, nous allons avoir le même résultat. Il faut plutôt considérer chaque ensemble disjoint comme un seul sommet et réappliquer la même logique. Autrement dit, nous ne considérons plus les arêtes les moins coûteuses pour atteindre chaque sommet, nous nous intéressons maintenant aux arêtes les moins coûteuses qui vont connecter les groupes de sommets ensemble.

Cette étape est répétée jusqu'à ce que tous nos sommets ne soient plus que dans un seul groupe. Avec un nombre n de sommets, $log\ n$ étapes sont nécessaires, car à chaque étape il y aura au moins deux fois moins de groupes dans le graphe. Voici la logique générale pour l'algorithme de Borůvka.

Tant que le MST n'est pas terminé (tous les sommets sont dans un seul groupe)
{
- Trouver les meilleures arêtes pour chaque sommet ou groupe
- Fusionner les groupes (UNION-FIND)
- Enlever les arêtes utilisées et celles qui créent des cycles pour la prochaine itération
}

Algorithme 3 : Borůvka

1.5.1 Exemple

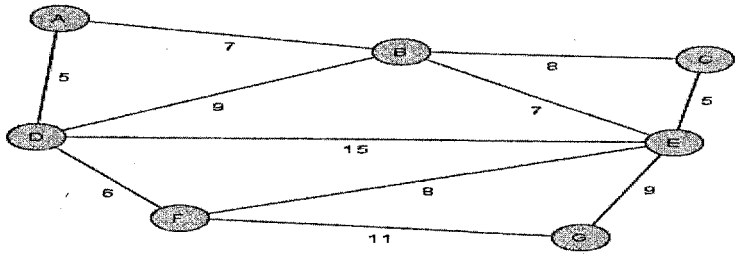


Figure 6 : État initial

La Figure 6 montre le graphe dans son état initial. Nous allons donc commencer par sélectionner les arêtes les plus courtes incidentes à chaque sommet (propriété de l'arête de poids minimum).

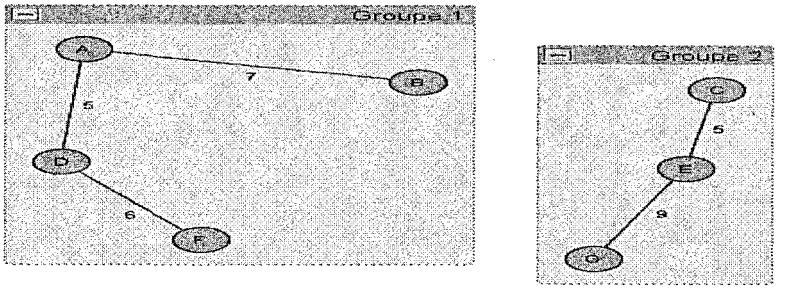


Figure 7 : Première étape

Dans la Figure 7, les sommets A et D ont donc sélectionné l'arête AD, le sommet B a sélectionné l'arête AB et ainsi de suite. Ce qui a créé deux ensembles disjoints dans le graphe. Nous appliquons maintenant une autre étape de Borůvka en sélectionnant les arêtes les plus courtes incidentes à ces deux groupes.

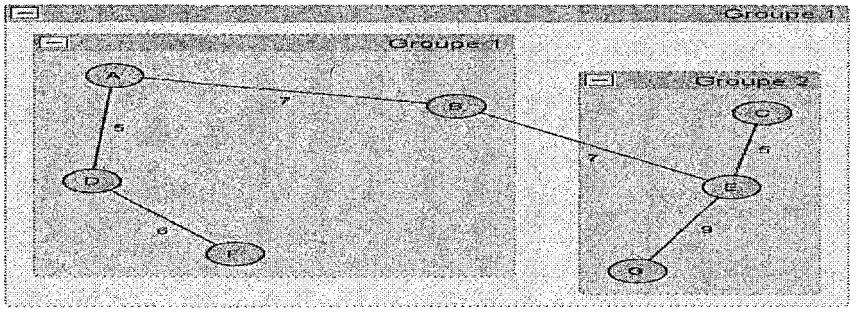


Figure 8 : Deuxième étape

Dans la Figure 8, nous avons choisi l'arête BE, car c'était l'arête la plus courte qui reliait le groupe 1 avec le groupe 2. Ces deux groupes sont donc fusionnés dans un seul groupe et nous choisissons arbitrairement que le groupe 2 soit le groupe qui se

fait assimiler dans le groupe 1. Quand tous les sommets ne sont que dans un seul groupe, l'algorithme se termine car le MST est trouvé.

Nous allons maintenant nous attarder au temps d'exécution de cet algorithme. Soit m le nombre d'arêtes et n le nombre de sommets. Une étape de Borůvka élimine au minimum la moitié des sommets et crée donc au plus n/2 groupes. Dans l'exemple, nous avons 2 groupes après la première étape de Borůvka. Cette situation est plutôt idéale pour le temps d'exécution, mais nous aurions pu avoir une situation différente avec par exemple AB dans un groupe et DF dans un autre groupe. Cette situation se produit quand les deux sommets ont choisi la même arête. Il faut néanmoins faire l'analyse du temps en pire cas.

Il faut donc O(lg n) étapes pour mettre tous les sommets dans le même groupe. Il faut visiter toutes les arêtes à chaque étape de Borůvka afin de trouver les arêtes minimales incidentes à chaque sommet et donc il y a un travail O(m) fait à chaque étape. Cependant, nous n'avons pas besoin de passer plusieurs fois sur des arêtes déjà utilisées. Nous pouvons éliminer les arêtes déjà utilisées en temps O(1) en utilisant une liste chainée ou alors en créant un nouveau tableau avec les arêtes non utilisées pour chaque étape de l'algorithme. Les arêtes non utilisées sont les arêtes qui ne créent pas de cycle et les arêtes uniques. Dans ce dernier cas, un espace mémoire de O(m) est suffisant, car il suffit de réécrire dans le même tableau à chaque étape ou alors effacer une arête dans une liste chainée. Karger, Klein et Tarjan [Karger, Klein, Tarjan 1995] ont démontré qu'avec une telle technique le temps total pour visiter

toutes ces arêtes est réduit d'un facteur de $O(n^2)$.La version sans contraction nécessite également un petit travail de l'algorithme UNION-FIND pour savoir dans quel groupe appartient quel sommet. En utilisant la compression par chemin de Tarjan [Tarjan 1975], nous ajoutons donc un temps $\alpha(m)$ à chaque étape. Par conséquent, la complexité temporelle dans le pire cas est en $O(\min(m\,\alpha(m)\log n, n^2)$.

1.6 *Algorithme de Kruskal*

L'algorithme de Kruskal est considéré comme l'algorithme le plus simple à comprendre et à implémenter. Kruskal conçu cet algorithme suite à la lecture du papier de Borůvka en 1956. Contrairement à l'algorithme de Borůvka et celui de Prim, l'algorithme de Kruskal nécessite auparavant le tri des arêtes. Cette condition permet de simplifier grandement le fonctionnement de l'algorithme. Il ne s'agit en fait que de placer les meilleures arêtes les unes après les autres, tant qu'elles ne créent pas de cycle. L'algorithme suivant représente les étapes nécessaires que fait l'algorithme de Kruskal, sans entrer dans la complexité temporelle des algorithmes d'UNION-FIND et de tri.

Trier les Arêtes en ordre croissant

Répéter{
Ajouter l'arête de poids minimum parmi les arêtes disponibles.
- *Si l'arête ajoutée forme un cycle, recommencer avec la prochaine arête.*
- *S'il ne reste plus d'arêtes, le MST est terminé avec succès.*
- *Si tous les sommets sont atteints par des arêtes et que tous les sommets sont dans un seul ensemble disjoint, l'algorithme arrête, car le MST est terminé.*

}
Fin répéter

Algorithme 4 : Kruskal

L'algorithme de Kruskal se fait en deux phases. La première phase fait un tri des arêtes et la deuxième phase construit l'arbre couvrant minimal en plaçant les plus petites arêtes tant qu'elles ne créent pas de cycle. Le tri des arêtes prend un temps O(m lg m) pour un problème avec m arêtes. La boucle principale est exécutée dans le pire des cas m fois quand le graphe est creux (toutes les arêtes sont nécessaires pour faire le MST). Et nous avons également affaire au pire cas quand l'arête la plus coûteuse est nécessaire pour faire le MST. Cependant, dans un problème typique avec un graphe très dense, il faut s'attendre à ce que la boucle principale ne s'exécute qu'une fraction de fois car l'algorithme s'arrêtera avant. En effet, si une certaine condition est respectée, l'algorithme peut se terminer prématurément.

Cette condition est la suivante : si les sommets sont visités et que tous les sommets sont dans le même groupe, alors le MST est trouvé. Donc, la boucle principale va s'exécuter un total de m fois. Chaque arête considérée doit être testée pour voir si cette dernière crée un cycle. L'algorithme d'UNION-FIND [Tarjan 1975] est donc utilisé pour détecter les cycles. Cette structure ajoute un coût de $\alpha(m)$ opérations par itération. Si nous nous intéressons maintenant à la complexité temporelle de l'algorithme de Kruskal, nous devons additionner le temps du tri avec le temps de la boucle principale. Les meilleurs algorithmes de tri par comparaison (Quicksort) ont un temps d'exécution de O(m lg m) et la boucle principale a un temps d'exécution en $O(m (\alpha(m)))$. Nous avons donc un temps total égal à O(m log m). Le tri domine donc le temps de cet algorithme. Pour accélérer l'algorithme de Kruskal, il faut

donc utiliser un bon algorithme de tri, un bon algorithme d'UNION-FIND et implémenter la condition d'arrêt.

1.7 *Algorithme de Prim*

L'algorithme de Prim [Prim 1957] est un algorithme qui trouve l'arbre couvrant minimal d'une manière plus centralisée. En effet, à chaque itération de l'algorithme, l'élément le plus proche de l'arbre courant est ajouté. Si un graphe possède un nombre S de sommets, il faudra $(S-1)$ itérations pour couvrir tous les sommets du graphe et obtenir l'arbre couvrant minimal.

L'algorithme de Prim a donc un fonctionnement très simple. L'ensemble de départ contient un sommet $S_i \in S$ et à chaque itération la meilleure arête est choisie pour étendre notre ensemble de façon optimale. Dans le cas du problème du MST, nous choisissons la plus courte arête qui relie notre ensemble à un nouveau sommet. Nous pouvons donc découper les différentes tâches de Prim comme suit :

1. *Création de l'ensemble de départ.*
2. *Sélection de la meilleure arête à ajouter pour agrandir notre ensemble de départ.*

L'étape 1 ne peut pas être optimisée. La création de l'ensemble de départ se fait toujours en temps constant en choisissant aléatoirement ou arbitrairement le sommet de départ. L'étape 2 doit être répétée $(S-1)$ fois pour trouver toutes les arêtes du MST. En revanche, le processus de sélection peut être amélioré à l'aide d'une meilleure structure de données. Le but de ce chapitre est donc de montrer dans un

premier temps comment fonctionne l'algorithme de Prim et, dans un deuxième temps, de montrer comment améliorer son temps d'exécution avec de meilleures structures de données.

1.8 *Exemple étape par étape*

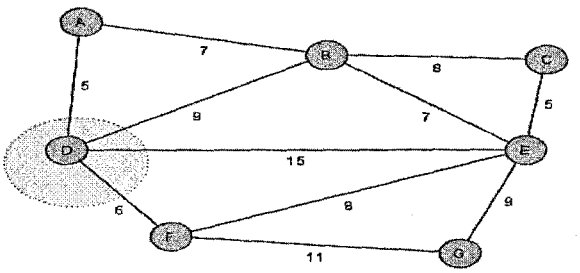


Figure 9: MST initial.

Le sommet D a été choisi comme sommet initial. Nous devons maintenant choisir une arête de D pour se rendre au prochain sommet : la plus courte est l'arête DA.

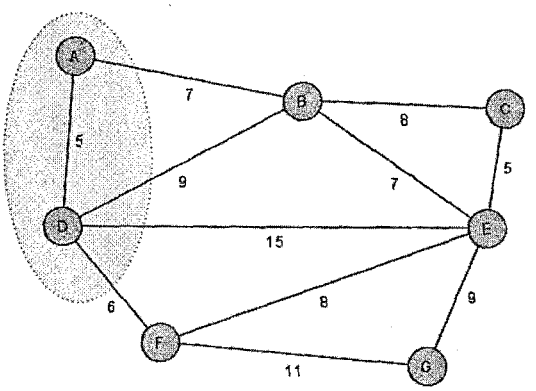


Figure 10 : Étape 1- agrandissement du MST avec l'arête DA

Quelle arête est la plus proche ? Nous choisissons l'arête DF pour étendre le MST.

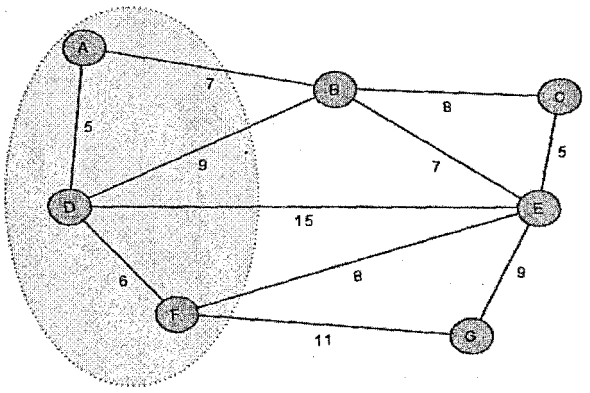


Figure 11 : Étape 2 - agrandissement du MST avec l'arête DF

Nous choisissons l'arête AB pour étendre le MST.

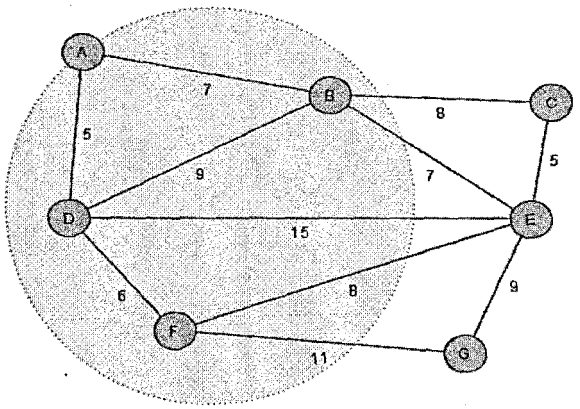


Figure 12 : Étape 3 - agrandissement du MST avec l'arête AB

Nous choisissons l'arête BE pour étendre le MST.

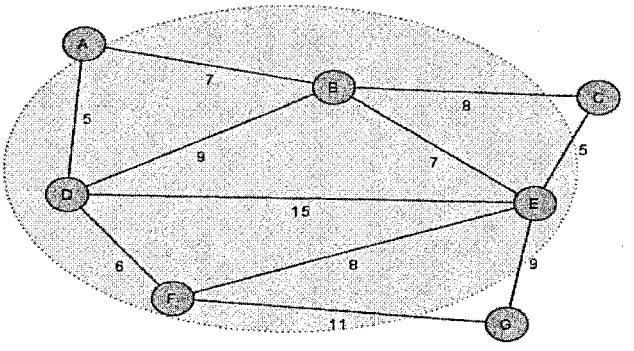


Figure 13 : Étape 4 - agrandissement du MST avec l'arête BE

Nous choisissons l'arête EC pour étendre le MST.

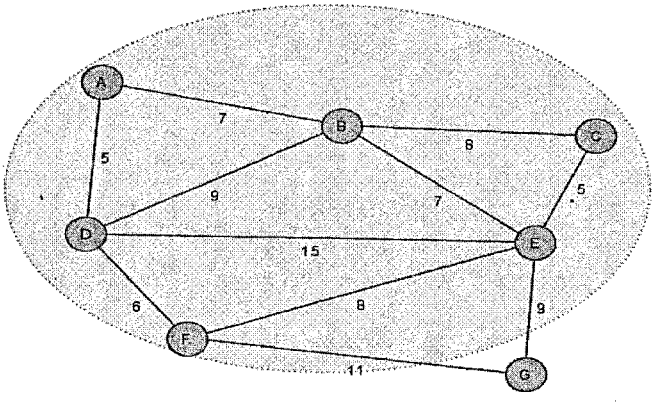


Figure 14 : Étape 5 - agrandissement du MST avec l'arête EC

Nous choisissons l'arête EG pour étendre le MST.

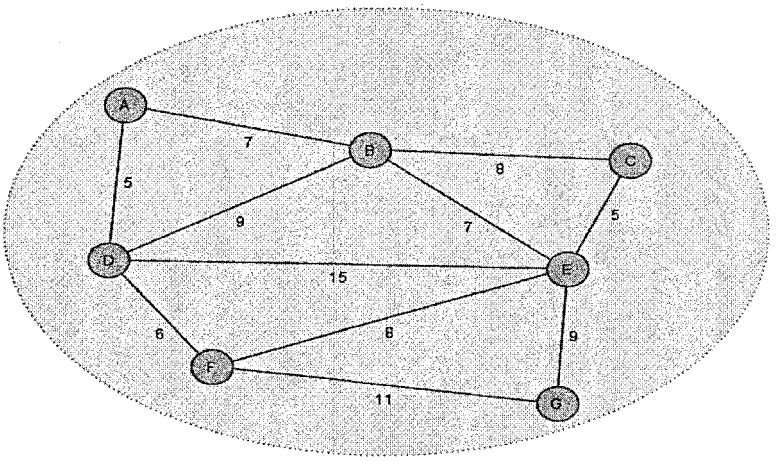


Figure 15 : Étape 6 - agrandissement du MST avec l'arête EG

Tous les sommets ont été rejoints, l'arbre couvrant minimal est trouvé.

1.9 *Algorithme de Prim avec une matrice de distances*

La matrice d'adjacence est une structure de donnée assez générale pour représenter un graphe. Elle est définie comme suit : si la case *ij* de la matrice possède la valeur 1, cela veut dire qu'une arête existe entre le sommet i et le sommet j.

La matrice de distances inscrit la distance de l'arête dans la case ij. Par exemple, le chiffre 5 indique que l'arête a un poids de 5 alors qu'une valeur négative indique l'absence d'une arête. Dans un graphe très dense, chaque sommet possède une arête vers tous les autres sommets du graphe, ce qui nécessite donc une matrice de taille n^2 pour représenter toutes les arêtes possibles.

L'algorithme de Prim avec une matrice de distances fonctionne en répétant deux opérations dans la boucle principale : l'ajustement des priorités et le choix du prochain sommet. L'ajustement des priorités se fait en itérant sur toute la ligne de la matrice correspondant au sommet courant. Pendant que cette ligne de matrice est visitée, un vecteur de priorités contenant la proximité de chaque sommet est ajusté. Ensuite, le choix du prochain sommet se fait en itérant sur tout le vecteur de priorités ajusté précédemment et en choisissant l'arête la plus courte. À partir de cette arête, un nouveau sommet est atteint l'algorithme recommence l'ajustement des priorités. L'algorithme se termine quand tous les sommets sont dans le MST, après n-1 étapes. Puisque les deux opérations ajustement et sélection fonctionnent respectivement en $O(n)$ et que l'algorithme les répète n-1 fois, l'algorithme de Prim avec la matrice de distances a une complexité temporelle en $O(n^2)$.

1.10 *Liste d'adjacence*

Les listes d'adjacence offrent un autre moyen de représenter les arêtes dans le graphe. Avec cette technique, pour chaque sommet $S_i \in S$, nous gardons une liste chainée contenant les sommets adjacents. Au niveau de l'implémentation, nous vérifions ensuite le poids de cette arête à l'aide d'un tableau externe ou alors nous encapsulons le poids de l'arête dans un objet.

Lors de la recherche, les listes chainées de tous les sommets présents dans l'ensemble de départ sont parcourues. Quand une arête est choisie, nous pouvons l'enlever des deux listes de sommets afin de ne plus la visiter lors des autres

itérations, ce qui sauve du temps. Les listes d'adjacence sont intéressantes pour les graphes moins denses qui ont plus d'arêtes que de sommets.

Nous allons maintenant parler des files de priorités qui offrent une solution très intéressante pour l'extraction de la meilleure arête. Toutes les files de priorité ont un certain ordre, ce qui permet d'effectuer la plupart des opérations importantes en un temps logarithmique.

CHAPITRE 2: LES FILES DE PRIORITÉ

2.1 *Introduction*

Une file de priorité est une structure de données qui permet d'extraire l'élément minimal d'un ensemble. Les files de priorité sont très importantes en informatique pour l'ordonnancement des processus au niveau du système d'exploitation, pour le tri externe de données et pour tous les algorithmes gloutons qui doivent faire le meilleur choix à chaque itération. L'algorithme de Prim est un de ces algorithmes.

Avant d'introduire les différentes files de priorité, il est bon de préciser que la file de priorité est avantageuse par rapport au tri complet dans le cas du problème de l'arbre couvrant minimal. En effet, le problème avec le tri complet des arêtes est que le problème ne nécessite pas autant de précision dans l'ordre des arêtes. Il est plus important de connaitre quelle est la meilleure arête à utiliser plutôt que de trier des arêtes qui ne seront peut-être même pas utilisées. Car il faut en effet se souvenir car malgré qu'avec la densité, la taille du problème augmente, la taille du MST n'augmente pas car celle-ci est toujours dépendante uniquement du nombre de sommets.

Dans ce chapitre, nous allons présenter plusieurs façons d'implémenter une file de priorité : la liste chaînée, l'arbre binaire de recherche, l'arbre équilibré, le tas binaire, le tas binomial et finalement le tas de Fibonacci.

Chaque file de priorité supporte deux opérations élémentaires : l'insertion d'un élément dans la file et l'extraction du minimum. Nous nommerons ces deux opérations

insert et extract-min. Il y a également une autre opération importante pour l'algorithme de Prim, c'est l'opération decreasekey. Cette opération consiste à diminuer la priorité d'un élément dans un tas ou une file de priorité.

2.2 *Liste chainée*

La première méthode pour implémenter la file de priorité est la liste chaînée. Nous pouvons utiliser deux stratégies lorsque nous utilisons la liste chaînée. Nous pouvons décider d'avoir une insertion rapide en O(1) et un extract-min lent en O(n) si nous ne gardons pas la liste triée. Si nous gardons la liste constamment triée alors l'extraction prend un temps O(1) et l'insertion prend un temps O(n), sans compter le temps de O(n lg n) pour trier les éléments initialement.

Cette implémentation peut également être utilisée avec un tableau plutôt que les pointeurs. L'avantage de la représentation par pointeurs (Figure 16) est de pouvoir éliminer efficacement des éléments de la liste.

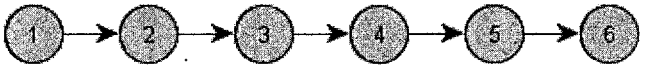


Figure 16 : Liste chainée

Voici un résumé des temps pour la liste chainée :

	Liste chainée triée		*Liste chainée non-triée*	
	Temps moyen	*Pire cas*	*Temps moyen*	*Pire cas*
Extract-min	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Insert	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Build-list	$O(n \log n)$	$O(n^2)$	$O(n)$	$O(n)$

Tableau 1 : Listes chainées

2.3 *Les tas*

Les tas sont des ensembles d'éléments qui ont une structure et un ordre. Ce sont des structures de données faites pour être des files de priorité performantes. Ils ont été conçus pour optimiser le temps d'exécution des opérations suivantes : extract-min, insert, minimum, union, decreasekey et delete. Dans ce qui suit, nous allons parler des trois tas les plus connus. Chacun d'eux a ses avantages et ses inconvénients. Le tas binaire a l'avantage d'être simple à implémenter et d'offrir une borne logarithmique O (lg n) pour la plupart de ses opérations. Le tas binomial est plus difficile à implémenter mais s'avère plus utile lorsque plusieurs files de priorité doivent être fusionnées, car l'union se fait beaucoup plus rapidement. Finalement, le tas de Fibonacci est ultimement beaucoup plus intéressant que le tas binomial, car en temps amorti, l'opération Decreasekey se fait en $O(1)$. Ce qui permet de diminuer la complexité temporelle de l'algorithme de Prim.

Le tas de Fibonacci est également plus rapide à construire car en temps amorti, l'insertion se fait en $O(1)$ contrairement à $O(lg\ n)$ pour le tas binomial. Le tas de Fibonacci ressemble fondamentalement au tas binomial, mais sa structure est beaucoup plus relâchée, ce qui lui permet de condenser le gros du travail sur d'autres opérations et d'accélérer certaines opérations.

Dans cette section, nous détaillerons le fonctionnement du tas binaire, du tas binomial et enfin du tas de Fibonacci. À la fin de la section, nous montrons comment utiliser ces files de priorité avec l'algorithme de Prim.

2.3.1 Comparaison des opérations de base sur différents tas

Le tableau suivant compare la complexité temporelle des opérations courantes sur les tas.

Procédure	Tas binaire (pire cas)	Tas binomial (pire cas)	Tas de Fibonacci (temps amorti)
INSERT	O(log n)	O(log n)	O(1)
MIN	O(1)	O(log n)*	O(1)
EXTRACT-MIN	O(log n)	O(log n)	O(log n)
UNION	O(n)	O(log n)	O(1)
DECREASEKEY	O(log n)	O(log n)	O(1)
DELETE	O(log n)	O(log n)	O(log n)

* Peut être amélioré en O(1) si nous gardons un pointeur vers l'élément minimal.

Tableau 2 : Comparaison des opérations des différents tas

2.3.2 Le tas binaire

Un tas binaire possède deux propriétés importantes : la structure et l'ordre. Lorsque chaque opération effectuée sur le tas binaire se termine, elle s'assure que le tas respecte encore ces deux propriétés importantes.

2.3.2.1 Propriété de la structure

Un tas binaire est un arbre binaire qui est rempli de haut en bas. Chaque niveau est deux fois plus épais que le précédent, comme dans un système binaire de nombres.

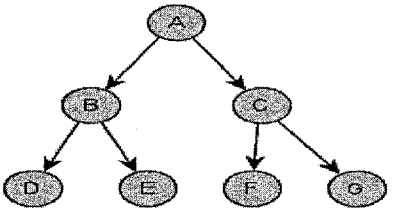


Figure 17 : Un tas binaire

Le tas binaire s'avère être une solution pratique pour implémenter une file de priorité, car la structure de donnée peut-être décrite à l'aide d'un simple tableau.

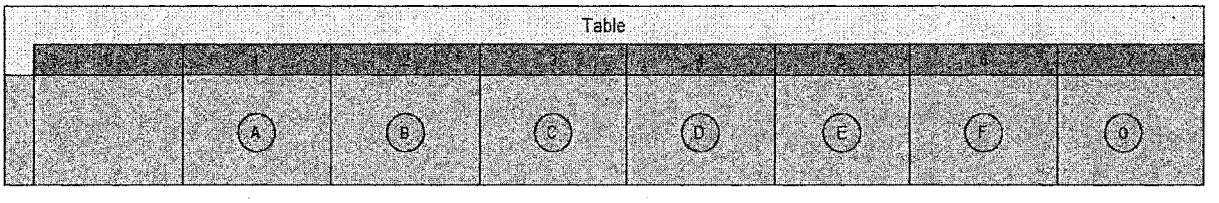


Figure 18 : Tas binaire dans un tableau

Dans une implémentation par tableau, l'arbre est décrit niveau par niveau. Le premier élément est toujours la racine et les enfants gauche et droit viennent tout juste après. Pour un élément à l'indice i du tableau, son enfant gauche sera toujours à

l'indice $2i$ et son enfant droit à $2i+1$. Le parent d'un élément à un indice i est à l'indice $\left\lfloor \frac{i}{2} \right\rfloor$.

Si nous commençons la racine à l'indice zéro, ces formules changent légèrement : l'enfant gauche est en position $(2i+1)$, l'enfant droit à $(2i+2)$ et le parent à $\left\lfloor \frac{i-1}{2} \right\rfloor$.

2.3.2.2 **Propriété de l'ordre**

La propriété de l'ordre implique tout simplement que le tas binaire doit être dans l'ordre après chaque opération. Un tas binaire qui s'occupe de classer les éléments par ordre croissant, également appelé tas-min a un ordre croissant à respecter.

Quand un nouvel élément est inséré dans le tas-min, il faut que cet élément soit remonté au niveau qui correspond à son ordre dans l'arbre. S'il est très petit, il sera remonté jusqu'à la racine. Sinon, il est possible qu'il remplacera un élément dans un autre niveau de l'arbre.

2.3.3 **Opération d'insertion**

L'insertion d'un nouvel élément se fait en ajoutant un élément à la prochaine position disponible dans l'arbre (comme une feuille) et en échangeant celui-ci avec son parent récursivement jusqu'à, possiblement, la racine. Cette opération prend le nom de percolation vers le haut.

Dans l'exemple de la Figure 19, l'élément 2 est ajouté dans le tas. A l'étape 1, nous voyons que l'élément 2 est ajouté comme une feuille de l'arbre. À l'étape 2, nous avons échangé celui-ci avec son parent. À l'étape 3, nous échangeons celui-ci avec le prochain parent. Et il n'y a pas d'étape 4, car le parent 1 est plus petit que 2. L'algorithme arrête.

Il est facile de voir que si l'élément ajouté se rend jusqu'à la racine, nous aurons fait $O(\log n)$ itérations d'échanges, car la hauteur de l'arbre est en $O(\log n)$ et que chaque opération de percolation augmente le niveau du nœud de 1.

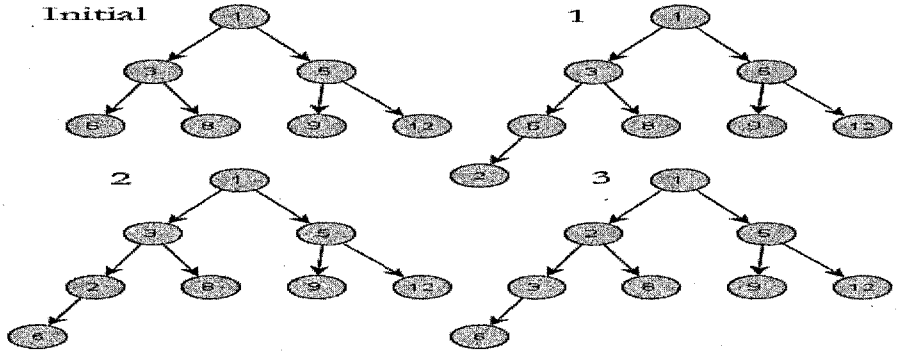


Figure 19 : Insertion d'un élément

2.3.4 **Suppression du minimum**

La suppression du minimum ou l'extraction du minimum utilise une percolation vers le bas. Le but est de bouger le dernier enfant de l'arbre dans l'espace vacant de la ligne qui a remplacé la racine. En procédant ainsi, l'arbre est assuré de garder son ordre et sa structure intacte.

Prenons l'exemple de la Figure 20. Nous colorons à l'étape 1 en rouge l'élément qui va combler l'espace vacant dans l'arbre. La racine est supprimée et les fils gauches et droits sont inspectés pour trouver le remplaçant. L'élément minimum est choisi (étape 2) et son ancienne position devient vacante. Le même procédé est répété à l'étape 3 et donc le fils gauche, 6 est remonté. L'espace vacant étant maintenant une feuille (pas de fils), nous déplaçons le 12 dans cet espace vacant et l'algorithme se termine.

La suppression du minimum s'effectue en $O(\log n)$. Il faut faire $O(\log n)$ percolations vers le bas pour remplacer la racine et ensuite nous faisons un travail constant pour déplacer le dernier enfant de l'arbre dans l'espace vacant, produisant un temps d'exécution en $O(\log n)$.

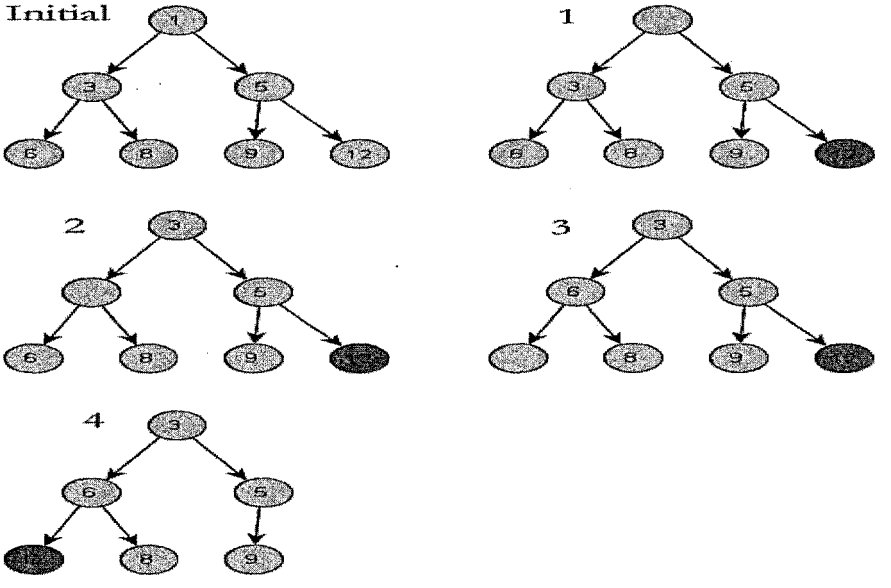


Figure 20 : Suppression du minimum

2.3.5 Opération decreasekey

L'opération decreasekey est facile à implémenter : il suffit de diminuer la valeur de la clé d'un nœud donné. Si l'opération a brisé l'ordre de l'arbre, le problème est réglé avec une série de percolations vers le haut. Sinon, rien n'est fait. Cette opération prend donc un temps O(lg n) dans le pire cas.

2.3.6 Opération delete

Pour supprimer un nœud donné qui n'est pas la racine, il faut tout simplement diminuer la clé de ce nœud au minimum afin qu'elle soit remontée à la racine. Et ensuite nous appliquons l'opération extract-min. Les deux opérations prennent chacune un temps en $O(lg\ n)$, ce qui nous donne donc un temps total de $O(lg\ n)$.

2.3.7 Construire le tas en temps linéaire

Créer un tas vide et faire n insertions prend un temps en $O(n\ lg\ n)$, car l'insertion prend un temps en $O(\log n)$.

Ce n'est pas la méthode optimale pour construire un tas à partir de zéro. Une meilleure méthode commence par remplir l'arbre sans se soucier de l'ordre des éléments, mais seulement de sa structure. Ensuite, pour faire respecter l'ordre, nous ferons une série de percolations vers le bas pour faire remonter les éléments minimums au sommet du tas. Cet algorithme est bien meilleur et il s'exécute en $O(n)$. Une preuve sera donnée après l'exemple.

2.3.8 Exemple de construction d'un tas

Nous reprenons les mêmes éléments utilisés dans le tas précédent. Ils ne sont pas triés. Nous les plaçons dans le tas consécutivement pour donner le tas initial de la Figure 21. Ensuite, nous allons faire des percolations vers le bas avec tous les éléments qui ne sont pas des feuilles. Dans cet exemple nous ferons donc 3 percolations vers le bas. Ces trois percolations feront en pire cas 2+1+1 = 4 échanges (les hauteurs des nœuds non-feuille sont additionnées pour atteindre le pire cas).

Figure 21 : Éléments à placer dans le tas

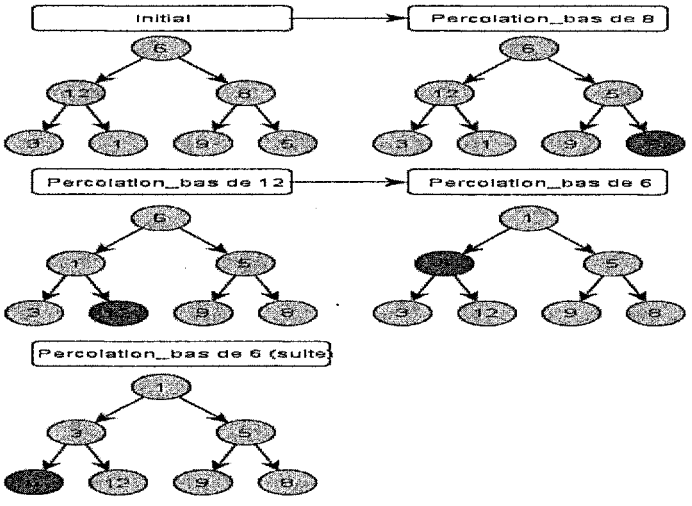


Figure 22 : Construction du tas

2.3.9 **Preuve de la borne**

Nous appliquerons cette série de percolations vers le bas à tous les éléments non-feuille de l'arbre. Nous pouvons prouver de plusieurs façons que le temps d'exécution de cet algorithme est en $O(n)$. Dans le pire cas, il faudra faire une percolation complète pour chaque élément non-feuille du tas. Par percolation complète, il faut comprendre que l'élément à la hauteur k va devoir descendre "$k$fois". Dans la Figure 22, la hauteur de la racine est de 2, et donc dans le pire cas, nous allons devoir descendre la racine de deux niveaux pour que l'ordre du tas soit respecté. Cette percolation complète était le pire cas pour un élément. Pour avoir le pire cas pour tous les éléments, c'est la somme des hauteurs de tous les nœuds de l'arbre.

Vu que la somme des hauteurs représente le nombre d'opérations en pire cas à effectuer pour construire le tas, si nous pouvons dire que ce nombre d'opérations est similaire au nombre de nœuds dans l'arbre, nous pouvons dire que cet algorithme s'exécute en temps linéaire, c'est-à-dire en $O(n)$.

Pour prouver que c'est le cas, nous pouvons faire quelques observations. Le nombre de nœuds augmente de la façon suivante : à chaque fois que nous ajoutons un nouveau niveau, ce niveau est deux fois plus grand que le précédent. Du côté des hauteurs maintenant, quand un nouveau niveau est rajouté, toutes les hauteurs précédentes sont augmentées de 1. Puisque seuls les nœuds non-feuille ont des hauteurs qui comptent, nous pouvons ajouter la taille de l'arbre précédent à chaque itération puisque chacune de ces feuilles va prendre de la hauteur. Nous pouvons le

voir sur la Figure 23 à hauteur de 2, il suffit de prendre la somme des hauteurs précédentes et de lui ajouter 3 (l'arbre précédent) pour obtenir 4, la nouvelle somme des hauteurs. La valeur de la somme des hauteurs est donc totalement dépendante du nombre de nœuds pour sa croissance. Ce qui prouve que ces deux quantités sont dans le même ordre et donc l'algorithme construire tas s'effectue en $O(n)$.

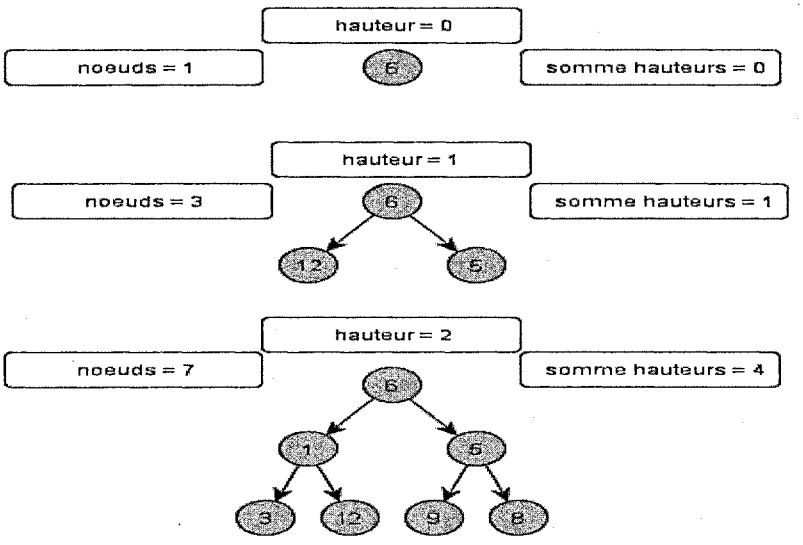


Figure 23 : Nombre de nœuds et somme des hauteurs

2.4 *Le tas binomial*

Un tas binomial [Vuillemin 1978] est une liste chaînée d'arbres ordonnés par rangs. Les rangs vont de 0 à k et représentent la taille de chaque arbre. Un arbre de taille k a 2^k nœuds au total et l'arbre $(k + 1)$ a toujours deux fois plus de nœuds que le précédent. Ce qui fait des tas binomiaux une structure ordonnée prévisible et parfaite pour améliorer la vitesse d'une opération importante : l'union.

Un arbre binomial de taille 0 est un simple nœud alors qu'un arbre binomial de taille 8 possède $2^8 = 256$ nœuds. La Figure 24 montre la structure des arbres binomiaux.

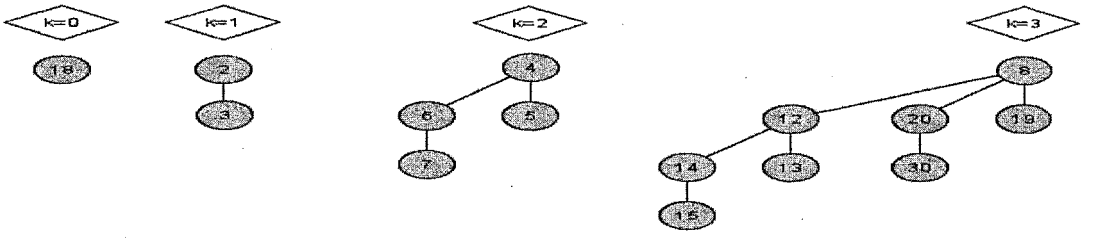


Figure 24 : arbres binomiaux de degré k

Un tas binomial est une file de priorité constituée d'arbres binomiaux. Chaque tas binomial peut avoir au maximum $O(lg\ n)$ arbres binomiaux si le nombre total de valeurs est n.

2.4.1 **Structure du nœud binomial**

Le tas binomial est une liste d'arbres binomiaux (Figure 25) et ces arbres binomiaux sont une collection de nœuds binomiaux. Le tas est donc un ensemble de nœuds binomiaux. Chaque nœud binomial peut avoir un parent, un fils et un frère. Le pointeur frère sera utilisé au niveau de la racine pour parcourir les arbres binomiaux de degrés k. Les pointeurs enfant et frère sont utilisés pour créer des arbres de degré k et le pointeur parent permet l'implémentation des opérations delete et decreaseKey en temps logarithmique. Quand un nœud n'a pas de fils, de parent ou de frère, la valeur du pointeur est à NIL

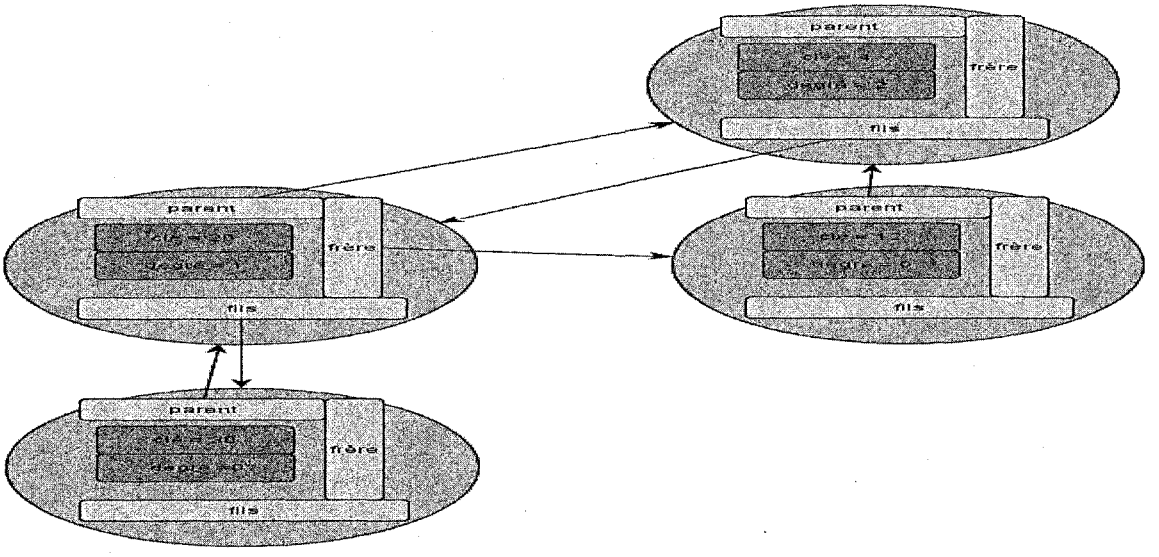


Figure 25 : Arbre binomial

2.4.2 Trouver le minimum

L'élément minimal d'un tas binomial peut-être trouvé en $O(\log n)$ temps. Dans un tas binomial, la racine de chacun des arbres k peut être l'élément minimum. Il faut donc parcourir toutes les racines pour trouver la plus petite. Puisqu'il y a au maximum $O(\log n)$ arbres, trouver la valeur minimale nécessite un temps en $O(\log n)$.

En revanche, si nous gardons un pointeur vers la racine minimale pendant toutes nos opérations, alors nous pouvons trouver l'élément minimum en temps constant. Ajouter ce pointeur ajoute un peu plus de complexité spatiale à l'implémentation du tas binomial car il faut gérer un pointeur pour chaque tas binomial, ce qui vient nuire à l'idée qu'un tas binomial est uniquement une collection de nœuds binomiaux. L'algorithme est donné dans [Cormen, Leiserson, Rivest and Stein 2009]

```
TrouverMinimum(heap){
        min = heap;
        TANT QUE heap != NIL
          SI heap->clé < min->clé
                min = heap;
        heap = heap->frère;
        return min;
} //fin de la fonction
```

Algorithme 5 : TrouverMinimum

2.4.3 Union de deux tas binomiaux

La plus importante opération pour le tas binomial est l'opération de l'union. Cette opération utilise deux sous-programmes et participe dans ces trois opérations : l'insertion, l'extraction du minimum et la suppression. Toutes les opérations sauf celle de decreasekey vont utiliser l'union. Le premier sous-programme que nous allons examiner est l'union de deux arbres binomiaux de même degré.

2.4.4 Union de deux arbres de degré k

Pour unir deux arbres **a** et **b** de degré k, il faut déterminer d'abord quel arbre **a** la plus petite racine. Si l'arbre **a** est plus petit, il devient la racine du nouvel arbre. Il prend comme fils l'arbre **b** et l'arbre **b** le prend comme parent. L'arbre **b** prend également l'ancien fils de **a** comme frère.

```
UnionArbre(parent, enfant) {
        SI parent->clé > enfant->clé
                UnionArbre(enfant,parent) ;
        SINON
                enfant->parent = parent ; enfant->frere = parent->fils ;
                parent->fils = enfant ; parent->degré++ ;
} //fin de la fonction
```

Algorithme 6 : UnionArbre

2.4.5 **Union rapide**

Le deuxième sous-programme utilisé par l'union s'occupe de faire l'union naïve de deux tas binomiaux. Nous prenons les deux tas binomiaux et nous plaçons toutes leurs racines en ordre croissant de leur degré.

Le problème est bien entendu qu'un tas binomial ne peut avoir qu'un seul arbre de degré k. Avec l'union rapide, il y aura fort probablement des arbres de même degré qui seront côte à côte. C'est l'algorithme général de l'union qui s'occupera de régler ces conflits. L'algorithme provient également de [Cormen, Leiserson, Rivest and Stein 2009].

```
UnionRapide(heap1, heap2)
{
        newheap, newheapTete;
        SI heap1->degré < heap2->degré
          {
                newheap = heap1;
                heap1 = heap1->frère;
          }//fin du si
        SINON {
                 newheap = heap2;
                heap2 = heap2->frère;
                }
                newheapTete = newheap;

                TANT QUE heap1 != NIL ET heap2 != NIL
                {
                   SI heap1->degré < heap2->degré {
                          newheap->frere = heap1;
                          heap1 = heap1->frère;
                          newheap = newheap->frère;
                          }
                  SINON {
                            newheap->frere = heap2;
                            heap2 = heap2->frère;
                            newheap = newheap->frère;
                           }
                }// fin du tant que
```

```
        TANT QUE heap1 != NIL {
            newheap->frere =  heap1;
            heap1 = heap1->frère;
            newheap = newheap->frère;
          }//fin du tant que

        TANT QUE heap2 != NIL {
            newheap->frere =  heap2;
            heap2 = heap2->frère;
            newheap = newheap->frère;
          }// fin du tant que

        return newheapTete;
} //fin de la fonction
```

Algorithme 7 : Union Rapide

La fonction UnionRapide prend en paramètre deux tas binomiaux. Il faut déjà commencer par déterminer quel sera le nœud racine qui sera la tête de la nouvelle liste. Ensuite, tant que les deux tas ont des arbres à placer, nous mettons les arbres en ordre croissant de leur degré dans le nouveau tas.

Éventuellement, cette boucle s'arrête quand un des tas devient vide. Après cela, nous déchargeons les éléments restants de ce tas dans le nouveau tas binomial. Et nous retournons le pointeur vers le nouveau tas (qui est un pointeur sur le premier élément du tas). Les premières lignes de l'algorithme font le même travail que la boucle interne, mais il est important de traiter la première itération différemment afin de pouvoir préserver le pointeur sur le bon nœud.

2.4.6 **Opération d'UNION**

L'opération de l'union remplit une fonction supplémentaire pour le tas binomial : l'insertion. En effet, pour insérer un nouvel élément, il suffit de faire l'union d'un tas contenant seulement l'élément à insérer.

La procédure prend en paramètre deux tas. En premier nous faisons une union rapide des deux tas afin d'avoir les tas en ordre k, k+1, k+2 avec de possibles doublons. Ensuite, l'algorithme va itérer sur cette liste en utilisant trois pointeurs de position. Un pointeur avant, un pointeur courant et un pointeur après. Nous pouvons nommer ces pointeurs avant_x, x, et apres_x. Ensuite, en fonction des degrés des arbres de nos trois pointeurs, la boucle interne va agir avec quatre cas différents. Le but de cette boucle sera de régler tous les conflits de structure de la liste en gérant chaque situation en fonction des quatre cas programmés.

Pour comprendre les quatre cas de figure, nous allons utiliser un exemple d'exécution. Nous commençons avec deux tas binomiaux (Figure 26 et Figure 27). Nous concaténons ces deux tas avec l'opération UnionRapide, ce qui donne le résultat de la Figure 28. Ce tas ne respecte pas la structure, car plusieurs arbres de même degré existent. Il faudra donc fusionner des arbres k pour faire des arbres k+1.

Nos pointeurs commencent sur x->50 et après-x->20. Puisque les deux arbres ont le même degré, nous faisons une fusion avec l'opération UnionArbre. Nous faisons avancer les pointeurs et nous continuons à la deuxième itération. Le résultat de la

première itération peut être vu avec la Figure 29, le résultat de la deuxième itération avec la Figure 30 et le résultat de la dernière itération avec la Figure 31.

La deuxième itération nous présente un autre cas de figure : il y a trois arbres de même degré dans l'arbre. La solution à ce cas de figure est d'avancer les pointeurs pour que le premier arbre des trois, garde son degré et que les deux autres soient fusionnés. À la 3e itération, nous revenons dans un cas de base : deux arbres de même degré. Nous faisons la fusion et les pointeurs sont avancés. Ces fusions successives ont donné deux arbres de degré 2. Nous les fusionnons et les pointeurs sont avancés. Le pointeur frère est maintenant égal à NIL et l'algorithme arrête. Le tas résultant est retourné, soit le pointeur sur la première racine de la file.

Une bonne implémentation de l'union devra également gérer quelques détails de pointeurs. Par exemple, si nous fusionnons les deux premiers arbres ensemble, et que la deuxième racine devient le parent de la première racine, il faudra que le pointeur de file pointe directement sur la deuxième racine, car c'est le nouveau pointeur qui va représenter la file. L'algorithme provient du [Cormen, Leiserson, Rivest and Stein 2009].

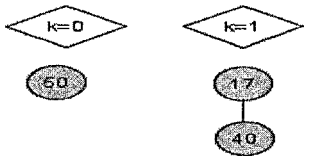


Figure 26 : Tas 1

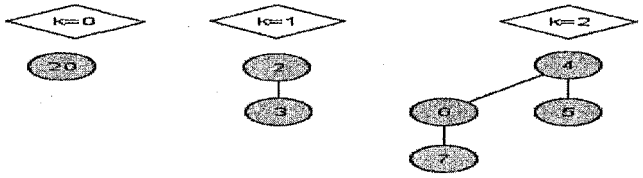


Figure 27 : Tas 2

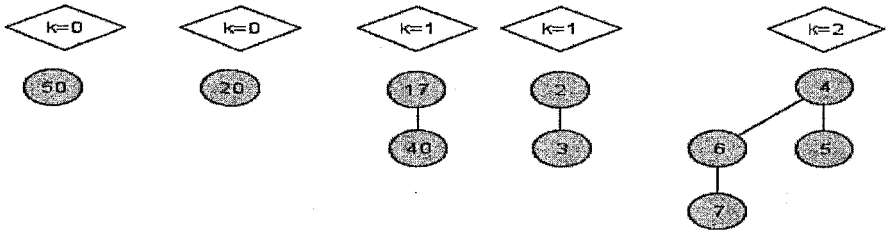


Figure 28 : Union rapide de tas 1 et tas 2

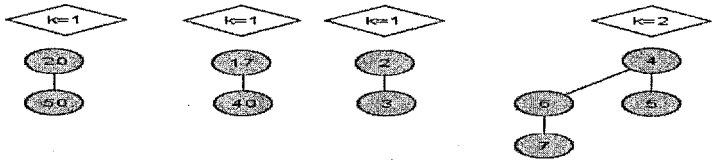


Figure 29 : Itération 1

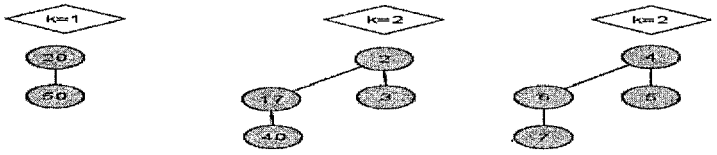


Figure 30 : Itérations 2 et 3

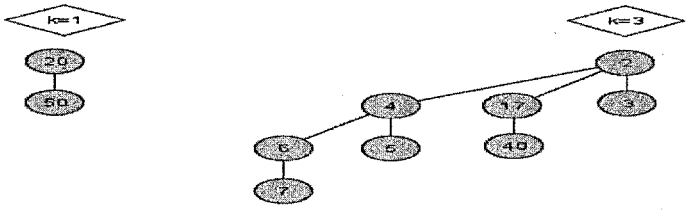


Figure 31 : Itération 4 et FIN

```
mergeHeap(heap1,heap2) {
  newheap = UnionRapide(heap1, heap2) ;
  SI newheap == NIL
        return newheap;

  x = NIL; //racine en cours d'examen
  apres_x = NIL; //racine apres x
  avant_x = NIL; //racine avant x
   x = newheap;
  apres_x = newheap->frere;

  TANT QUE apres_x != NIL)
   {
    SI (x->degre != apres_x->degre || apres_x->frere != NIL && apres_x->frere->degre == x->degre)
        {
          avant_x = x;
          x = apres_x;
        }
   SINON {
            SI x->clé <= apres_x->clé
              {
                 x->frere = apres_x->frere;
                 unionArbre(apres_x, x) ;
              }
         SINON {
                 SI avant_x == NIL
                    newheap = apres_x;
                 SINON
                    avant_x->frere = apres_x;
                 unionArbre(x, apres_x) ;
                 x = apres_x;
           } //fin du sinon
          apres_x = x->frere;
        }

    } //fin du tant que
  return newheap;
}//fin de la fonction
```

Algorithme 8 : MergeHeap

2.4.7 Opération DecreaseKey

L'opération DecreaseKey est très simple grâce au pointeur sur le parent. Il suffit de changer la valeur de la clé, vérifier le parent pour voir si l'ordre est encore respecté et, si l'ordre doit être changé, nous échangeons les nœuds dans l'arbre binomial. Cette

opération s'effectue en temps logarithmique. Il faut procéder par échange de valeur lorsque nous voulons permuter un nœud avec son parent, car si nous procédons par échange de pointeurs, l'algorithme DecreaseKey ne va pas fonctionner. Le problème réside dans la nature de la liste chaînée du tas binomial : c'est une liste simplement chaînée. Par exemple, si nous permutons un élément jusqu'à sa racine en changeant à chaque fois les pointeurs, l'élément que nous avons changé à la racine pointe sur un nœud frère à sa droite, mais n'a aucune connaissance d'un frère qui pointe sur lui à gauche. C'est un problème qui n'existe pas avec le tas de Fibonacci, car ce dernier utilise une liste doublement chaînée.

```
decreaseKey(element, valeur)
{
        element->key = valeur;
        i = element;
        SI (i->parent == NULL)
                return ;
        TANT QUE (i->parent != NULL)
        {
                cléParent = i->parent->key;
                SI (cléParent > i->key )
                   swap_parent_child(i, i->parent);
                SINON break
                i = i->parent;
        }
}//fin de la fonction
```

Algorithme 9 : DecreaseKey

2.4.8 Suppression du minimum

La suppression du minimum s'effectue de la façon suivante : nous détruisons le nœud minimal, nous transformons sa liste d'enfants en tas binomial et nous faisons l'union des deux tas. Quand nous détruisons le nœud minimal, il faut procéder comme

pour un effacement dans une liste chainée : le nœud précédent est relié avec le nœud suivant. Puisque la structure des nœuds binomiaux n'a pas de pointeur vers le nœud précédent, il faudra itérer à partir du début des racines vers l'endroit voulu. Ce qui peut être accompli aisément grâce à l'information du degré de la racine que nous enlevons. Si nous enlevons une racine de degré 5, il suffit de faire avancer le pointeur à partir du début du tas jusqu'à tomber sur une racine de degré 5.

Ensuite, quand nous créons un nouveau tas binomial avec les enfants du nœud détruit, il faut réordonner cette liste d'un ordre décroissant de degrés à un ordre croissant de degrés. En d'autres mots, il faut inverser l'ordre des enfants. Tout en faisant cela, il faut changer le pointeur parent à NIL, sinon une future opération decreaseKey ne s'arrêtera jamais.

Une fois le nouveau tas binomial prêt (Figure 32), l'union est faite avec l'autre et le tas binomial résultat est retourné. Si nous voulons simuler une fonction extract-min, il faut d'abord faire appel à la fonction TrouverMinimum et ensuite appeler la fonction SuppressionMinimum.

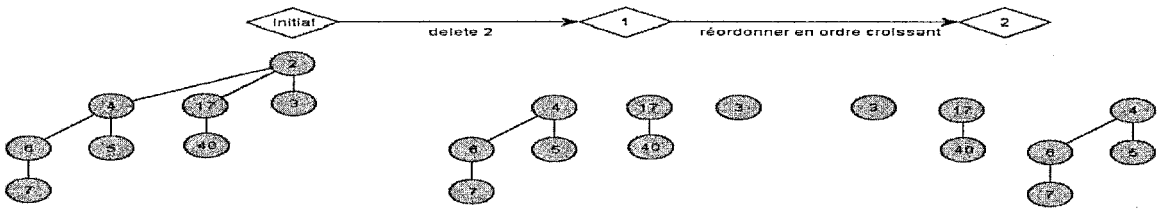


Figure 32 : Inversion des enfants

```
extract-min(heap){
        x = findMinimum(heap) ; avant_x = heap;
        //On trouve l'élément avant x grâce au rank de x
        POUR i=0; i < x->rank -2 ; i++
              avant_x = avant_x->frere;

        //On supprime x (le minimum) de la liste
        avant_x->frere = x->frere;

        //On fait une liste chainée inversée des fils de minimum
        fils = x->child;
        SI (fils == NIL){ //pas de fils on return tout de suite
                heap = heap->frere;
                return heap;
        }
        courant = NIL; precedent = NIL;
        frere = NIL; i = 0; courant = fils;
        TANT QUE courant != NIL{
             SI precedent > 0 {
                courant->parent = NIL; /Annuler le pointeur parent
                frere = courant->frere; //Le prochain élément de la liste
                courant->frere = precedent;
                precedent = courant;
                courant = frère;
              }
            SINON{
              precedent = courant;
              courant = courant->frere;
              precedent->frere = NIL; //,Car cet élément devient le dernier élément de la liste.
              precedent->parent = NIL;
            }
        }//fin du TANT QUE
        teteListeEnfants = precedent;
        resultat = mergeHeap( heap, teteListeEnfants) ;
        return resultat;
} // fin de la fonction
```

Algorithme 10 : extract-min

2.5 *Tas de Fibonacci*

Le tas de Fibonacci [Fredman and Tarjan 1987] est très similaire au tas binomial. Il a une structure un peu plus lourde, car cette dernière contient plus d'information. Avec un tas binomial, chaque opération utilisée remet le tas dans un

état parfait, tandis que le tas de Fibonacci introduit un fonctionnement très souple avec certaines opérations. Seules certaines opérations vont déclencher ce qui est appelé une consolidation du tas, un algorithme qui met le tas de Fibonacci dans un état ordonné. Toutes les autres opérations sont très rapides. Un exemple d'une telle opération est l'opération DecreaseKey qui passe d'un temps logarithmique à un temps constant amorti. Ce qui a pour effet de diminuer la complexité temporelle de l'algorithme de Prim.

2.5.1 **Opération de l'Union**

Avec le tas de Fibonacci, la structure en liste chainée circulaire à double pointeur nous permet de faire l'union en temps constant. Les racines n'ont pas besoin d'être en ordre de degrés et nous pouvons avoir plusieurs racines du même degré. Le tas de Fibonacci est une liste doublement chainée circulaire (Figure 33), il n'y a donc pas de pointeur sur le début de la liste, car la liste n'a pas de début ni de fin. Il faut toujours sauvegarder un pointeur sur le minimum dans le tas de Fibonacci. Ce que nous pouvons faire c'est d'utiliser le pointeur vers l'élément minimum pour représenter notre tas. Si nous procédons ainsi, le tas de Fibonacci reste un ensemble de nœuds de Fibonacci et nous n'avons pas besoin d'avoir une structure englobante. L'algorithme 11 montre l'implémentation.

Figure 33 : Tas de Fibonacci

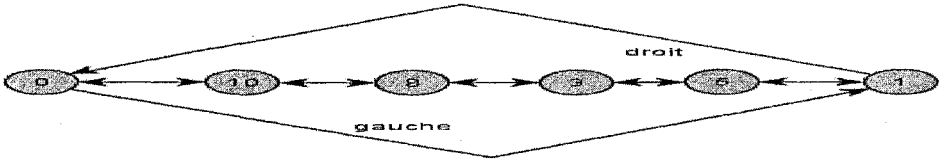


Figure 34 : Union des deux files

Ci-dessous le pseudocode pour l'union de deux files.

```
merge(a, b) {
        //Les trois cas pour éviter les pointeurs NIL
        SI (a == NIL && b == NIL)
                return NIL;
        SI (a == NIL && b != NIL)
                return b;
        SI (a != NIL && b == NIL)
                return a;
        //La file b est merged dans a
        //On change le minimum si celui de la nouvelle file est mieux
        SI (a->min->clé > b->min->clé)
                a->min = b->min;
        tmp = a->droit;
        a->droit = b->droit;
        a->droit->gauche = a;
        b->droit = tmp;
        b->droit->gauche = b;
        a->nodeCount += b->nodeCount;
        return a;
}// fin de la fonction
```

Algorithme 11 : Union Fibonacci

2.5.2 Opération d'insertion

L'insertion d'un nœud dans le tas se fait en temps constant. Nous créons un nouveau nœud et nous faisons l'union avec un autre tas. Puisque l'union se fait en temps constant, l'insertion se fait également en temps constant.

2.5.3 Extraction du minimum

L'extraction du minimum est l'opération la plus difficile à comprendre des tas de Fibonacci. Cette opération fait une consolidation des arbres semblable à l'union dans le tas binomial sauf qu'avec la structure plus complexe des arbres de Fibonacci, la consolidation devient elle aussi plus complexe. Afin de bien comprendre le fonctionnement de cette opération, nous allons utiliser un exemple et nous montrons comment évolue l'algorithme en images étape par étape. Notre exemple illustrera la première extraction du minimum sur un tas de Fibonacci tout juste créé.

Figure 35 : Tas initial

Voici le tas de Fibonacci initial (Figure 35). Ce dernier a été créé avec plusieurs insertions successives.

Nous nous débarrassons d'abord du minimum en changeant quelques pointeurs au niveau des racines. L'élément 0 va donc disparaitre.

Figure 36 : État initial

Ensuite tous les enfants du nœud minimum enlevé deviennent des racines. Dans l'exemple présent, le nœud 1 n'avait pas d'enfants donc rien ne se produit. Nous sommes maintenant rendus à l'étape de la consolidation. D'abord nous devons déclarer une table de pointeurs vers des arbres de degré k. Quand nous faisions la fusion des arbres binomiaux, nous n'avions pas besoin d'avoir une telle table, car les arbres de degré k étaient toujours côte à côte. Pour le tas de Fibonacci, nous avons besoin d'une table qui pointe vers les arbres de degré k, car ceux-ci ne sont pas tout le temps voisins, ils peuvent être partout dans la liste des racines.

Puisque le degré maximal d'un ensemble d'éléments est égal à son logarithme, nous n'avons besoin que d'une table de taille en $O(\log n)$.

Et puisque nous venons d'enlever le minimum de la liste et que le minimum était notre pointeur sur la liste, nous choisissons son frère droit comme nouveau pointeur de liste. Ce choix est complètement arbitraire et ne change pas l'algorithme. La Figure 37, Figure 38 et Figure 39 détaillent la consolidation dans le tas de Fibonacci.

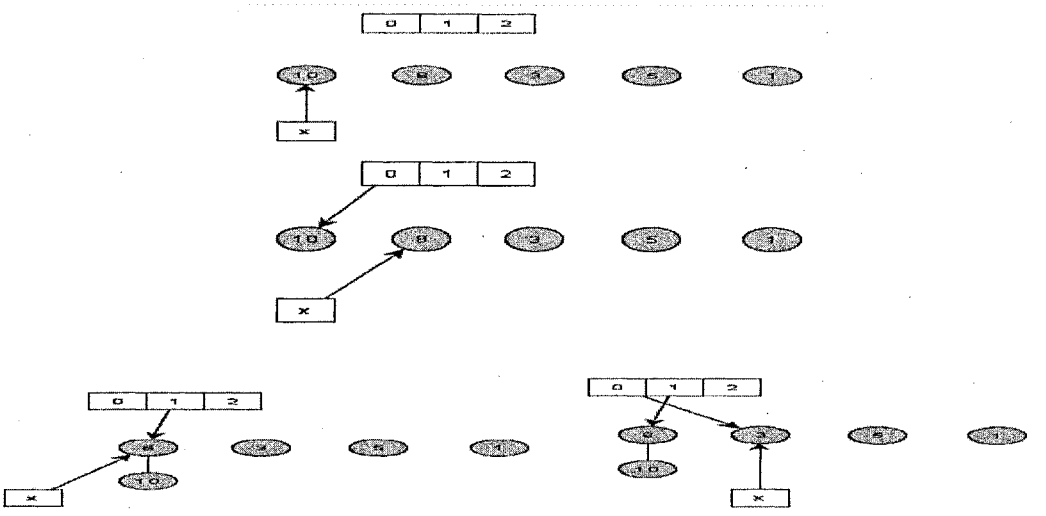


Figure 37 : étapes 1-4

Nous allons maintenant itérer sur la liste des racines. Notre première visite est sur le nœud 10 qui a un degré 0. Nous faisons pointer A[0] sur le nœud 10. La deuxième visite se fait sur le nœud 8 qui est lui aussi un degré 0. Puisqu'il y a un pointeur dans la table pour un arbre de ce degré, nous pouvons relier ces deux arbres (étape 3). Après avoir relié ces arbres, nous effaçons le pointeur dans la table, car ce nœud de degré 0 n'existe plus maintenant. Ensuite, puisqu'un nouvel arbre de degré 1 vient d'être créé, nous faisons pointer notre table directement sur celui-ci. Nous déplaçons notre pointeur sur le nœud 3 et nous enregistrons maintenant un nouveau nœud de degré 0 dans la table.

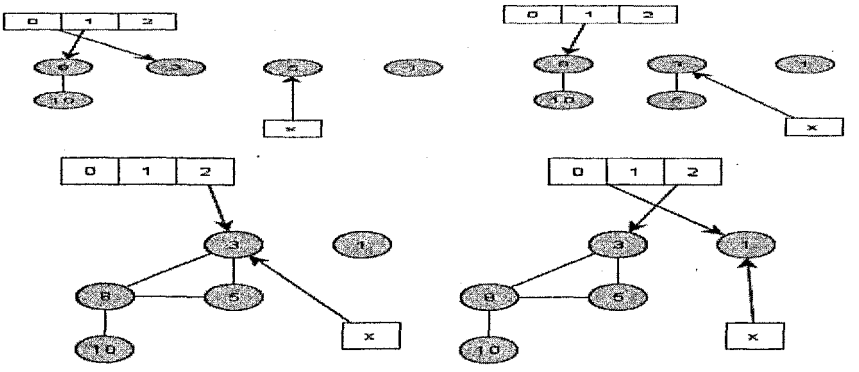


Figure 38: Étapes 5-9

Le pointeur se déplace à présent sur le nœud 5, qui est un nœud de degré 0. Puisque A[0] pointe sur 3, nous allons lier les racines 3 et 5 ensemble. À l'étape 6 maintenant, nous sommes sur une racine de degré 1 et nous avons A[1] qui pointe sur 8. Nous faisons une autre liaison (étape 8). À l'étape 9, nous pouvons voir que notre arbre ne pourra pas être consolidé plus que ça, la racine 1 ne pourra être liée avec aucune autre. Il faut cependant montrer comment l'algorithme s'arrête. Puisque nous ne pouvons pas savoir à l'avance le nombre exact d'itérations que prendra l'algorithme pour consolider tout le tas nous n'avons qu'à attendre que ce dernier nous propose de fusionner un nœud avec lui-même. À ce moment-là, nous sortons de l'algorithme de consolidation comme nous le montrons à la Figure 39.

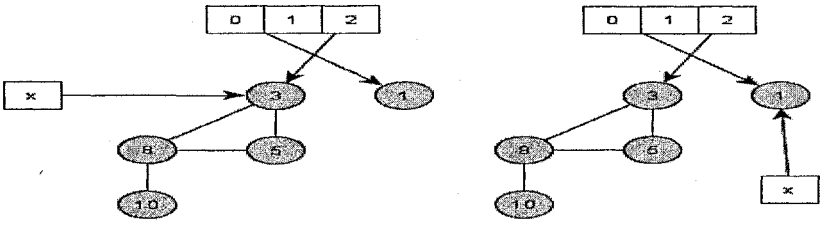


Figure 39 : Étapes 10-11 (Fin)

À l'étape 11, l'algorithme proposera de faire la fusion entre un x et un y sauf que x = y et donc nous arrêtons. Nous pouvons observer que la table et ses pointeurs correspond directement au chiffre 5 en binaire, 101, qui correspond également au nombre de nœuds.

Une fois la consolidation des racines faite, il faut trouver le nouveau nœud minimum. Il suffit d'itérer sur toutes les racines pour trouver le nouveau minimum. Il y a maintenant $O(\log n)$ racines, car nous venons de faire la consolidation. Cet algorithme offre les détails de l'implémentation de cette opération.

```
extract_min(heap){
        SI (heap == NIL)
           return NIL;
//On supprime le minimum
        min = heap->min; avant = min->left;
        apres = min->right; avant->right = apres;
        apres->left = avant;
//Le pointeur heap était peut-être le pointeur vers le minimum, on le change si c'est le cas
        SI (heap == min) {
            heap->right->nodeCount = heap->nodeCount;
            heap = heap->right;
        }
//Chaque fils du minimum devient une racine
         enfantCourant = min->child;

        POUR int i = 0; i < min->degree; i++
        {
                enfantCourant->parent = NIL;
                enfantCourant = enfantCourant->right;
        }
//On fusionne avec le heap courant comme ça les enfants deviennent des racines
        heap = merge(enfantCourant, heap) ;
        newmin = consolidate(heap) ;
        newmin->min = newmin;
        newmin->nodeCount = min->nodeCount - 1;

        return newmin;
} // fin de la fonction
```

```
consolidate(heap) {
        A[D[n]] //Initialisé à zéro, D[n] est le nombre de degrés possibles de noeuds
        depart = heap;
        x = heap; //itérateur
        y = 0; //pointeur sur arbre de degré d précédent
        d = 0; //degré de x
        RÉPÉTER
        {
                d = x->degree;
                TANT QUE (A[d] != NIL)
                {
                        y = A[d] ;

                //On a fini la consolidation
                        SI (x == y)
                           Aller à finConsolidation; //saut inconditionel vers finConsolidation
                        mergeTree(x, y) ;
                        A[d] = NIL;
                        d++;
                //Si y était le parent lors de la fusion, x devient y (pour l'instruction A[d] = x après)
                        SI (x->clé > y->clé)
                                x = y ;
                }
                A[d] = x;
                x = x->right; //pourrait etre déplacé dans la boucle à l'aide de la virgule
        }
        TANT QUE (x != depart) ;
 finConsolidation:

//Trouver le nouveau minimum
//On parcourt toutes les racines pour trouver le minimum

        depart = x;
        min = x;

        RÉPÉTER {
                SI (x->clé < min->clé)
                        min = x;
                x = x->right;
        }
        TANT QUE (x != depart) ;
        return min;
} // fin de la fonction
```

Algorithme 12 : ExtractMin

2.5.4 **Opération decreaseKey**

Le tas binomial et le tas binaire font l'opération decreaseKey en temps logarithmique, car ils font des échanges successifs avec le parent pour replacer le nœud au bon endroit une fois que sa valeur a changé. Rappelons-nous maintenant que le tas de Fibonacci n'impose pas de restrictions par rapport à ce qui peut être présent au niveau des racines. Nous pouvons avoir autant de racines que nous voulons, de n'importe quel degré, et le tas continuera de fonctionner normalement. Le tas de Fibonacci pourrait implémenter l'opération decreaseKey de la même façon que le tas binomial, mais cette flexibilité présente au niveau des racines permet une autre implémentation plus performante.

Dans le tas de Fibonacci, au lieu de faire remonter un nœud dans un arbre, nous allons directement prendre ce nœud et en faire une racine. Cette opération prend un temps constant. Cette opération introduit également le concept de "marquage" dans la structure de données. Lorsque nous coupons un nœud de son arbre, son parent se voit marqué, car il vient de perdre son enfant. Cette mesure a été introduite afin d'être capable d'enlever des nœuds dans un arbre de Fibonacci n' importe où et de pouvoir "purifier" la branche de l'arbre jusqu'à la racine. Cette purification est importante, car nous travaillons avec des arbres pouvant être fusionnés. Pour faire cette fusion, il faut que les arbres soient de même degré.

La Figure 40 montre plusieurs exemples de decreaseKey et ses conséquences sur un arbre de Fibonacci. Nous commençons avec notre arbre initial. Nous diminuons la valeur de la clé 7 à 1. Le nœud a maintenant une valeur plus petite que son nœud parent 6 et donc nous coupons ce nœud de l'arbre. Il devient une racine. Puisque ce nœud est également plus petit que le minimum courant, le minimum est mis à jour.

Nous répétons la même procédure avec le nœud 6. La clé est diminuée de 56, ce qui nous donne une clé de valeur -50. Cette clé est coupée de l'arbre et devient une racine. Nous mettons également le minimum à jour. Nous coupons cette fois le nœud 5 de la même façon. Cette fois, puisque son parent est marqué, il faut couper en cascade. La coupure en cascade fonctionne de la façon suivante : nous coupons tous les parents marqués jusqu'à la racine ou alors jusqu'à un parent non marqué. Si nous atteignons un parent non marqué, il est marqué et nous arrêtons. C'est un coupage récursif dans l'arbre. L'algorithme est présenté dans [Cormen, Leiserson, Rivest and Stein 2009].

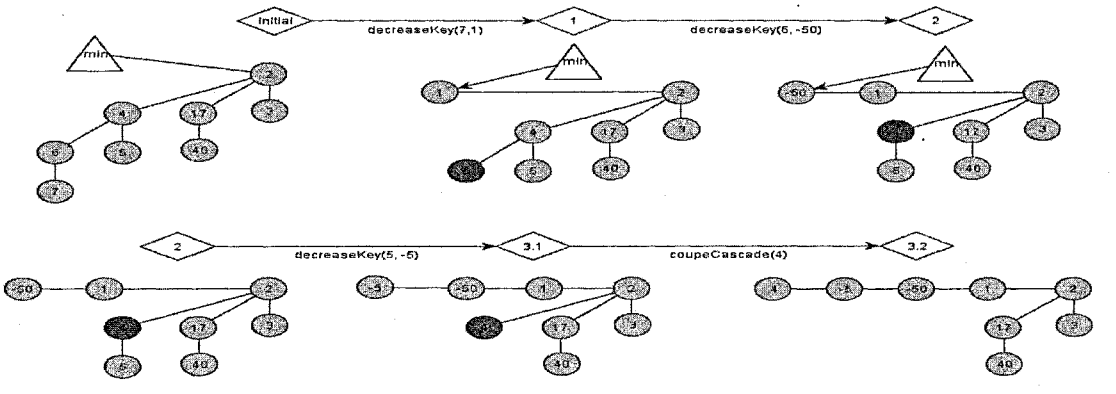


Figure 40 : Opération Decrease-key

```
decreaseKey(heap, x,  value) {
        x->key = newValue;
        y = x->parent;
        SI (y !=NULL && x->key < y->key) {
                Cut(heap, x, y);
                CutCascade(heap, y) ;
        }
        SI (x->key < heap->key)
             return x;
        return heap;
//y = parent
cutNode(heap, x, y) ;
//supprimer x de la liste des enfants de y en décrémentant y.degré
                y->degree--;
//Si l'enfant direct du parent est x, on doit changer l'enfant pour son frère ou NULL
                SI (y->child == x) {
                        //Si x a un frere, son frere devient le child de y
                        SI (x->left !=x) y->child = x->left;
                        //Si x n'a pas de frère, le child de y devient NULL
                        SINON SI (x->left == x) y->child = NULL;
                }

                //On doit modifier la liste de pointeurs dans laquelle x est
                avant = x->left;
                apres = x->right;
                avant->right = apres;
                apres->left = avant;

                x->left = x;
                x->right = x;
                x->parent = NULL; //x sera une racine donc pas de parent
                x->marked = false;//on démarque x

                //--- AJOUTER X À LA LISTE DES RACINES

                merge(heap, x) ;

cutCascade(heap, y){
        z = y->parent;
        SI (z != NULL) {
                SI (y->marked == false)
                        y->marked = true;
                SINON
                {
                        Cut(heap, y, z) ;
                        CutCascade(heap, z) ;
                }
        }
}\\ fin de la fonction
```

Algorithme 13 : decreaseKey

CHAPITRE 3: COMPARAISON EMPIRIQUE DES PERFORMANCES

3.1 *Introduction*

Dans ce chapitre, nous allons explorer l'avenue expérimentale de ces algorithmes. Quel algorithme est plus rapide sur une machine conventionnelle, et dans quels cas l'est-il ? Est-ce que l'analyse de la complexité temporelle concorde avec les tests que nous avons entrepris ?

Pour entreprendre une étude non biaisée des performances empiriques de ces algorithmes, nous allons tester ces derniers avec des graphes de tous les types, c'est-à-dire des graphes creux, des graphes moyennement denses et des graphes denses.

Pour pouvoir isoler la performance relative de certaines optimisations comme l'Union-Find, nous avons programmé plusieurs implémentations du même algorithme. Tous ces algorithmes ont été implémentés avec Visual C++ 2008 sur un ordinateur portable avec un processeur Intel Ivy Bridge i7-3610QM (3.3 Ghz) et 12 Go de mémoire vive. Les programmes ont été compilés en 32 bits et les tests prennent au maximum 4 Go de mémoire vive. La méthodologie de test utilisée ressemble à celle de Berenbaum [Berenbaum 1998]. Les graphes se distinguent sur deux aspects : le nombre de sommets et la densité. Berenbaum a fait son expérimentation sur des graphes allant de 100 à 1700 sommets et chaque graphe est testé avec 3 densités différentes : une densité de 0.2, 0.5 et 0.8. La densité p=0.2 signifie qu'il y a une

probabilité de 20% qu'il y ait une arête entre deux sommets distincts. Dans notre travail, nous testons des graphes allant jusqu'à 15000 sommets. Chaque taille de graphe est testée avec 3 densités différentes et 10 graphes aléatoires, pour un total de 450 tests. Puisque chaque test considère 12 algorithmes différents, 5400 résultats d'algorithmes sont ainsi générés.

3.2 *Les programmes testés*

Nous avons comparé tous les algorithmes dont le fonctionnement est décrit extensivement dans ce travail. Ci-dessous la liste des programmes testés. Il y a 4 versions de l'algorithme de Borůvka, 4 versions de l'algorithme de Prim et 4 versions de l'algorithme de Kruskal.

1. Algorithme de Borůvka avec gestion simple des ensembles disjoints
2. Algorithme de Borůvka avec compression de chemin récursive
3. Algorithme de Borůvka avec compression de chemin et union par rang, récursive
4. Algorithme de Borůvka avec compression de chemin et union par rang, itérative.
5. Algorithme de Prim avec matrice d'adjacence
6. Algorithme de Prim avec tas binaire
7. Algorithme de Prim avec tas binomial
8. Algorithme de Prim avec tas de Fibonacci
9. Algorithme de Kruskal avec gestion simple des ensembles disjoints
10. Algorithme de Kruskal avec gestion simple et condition d'arrêt
11. Algorithme de Kruskal avec compression de chemin
12. Algorithme de Kruskal avec compression de chemin et condition d'arrêt.

Sommets	Arêtes (densité 0.2)	Arêtes (densité 0.5)	Arêtes (densité 0.8)
1000	100000	250000	400000
2000	400000	1000000	1600000
3000	900000	2250000	3600000
4000	1600000	4000000	6400000
5000	2500000	6250000	10000000
6000	3600000	9000000	14400000
7000	4900000	12250000	19600000
8000	6400000	16000000	25600000
9000	8100000	20250000	32400000
10000	10000000	25000000	40000000
11000	12100000	30250000	48400000
12000	14400000	36000000	57600000
13000	16900000	42250000	67600000
14000	19600000	49000000	78400000
15000	22500000	56250000	90000000

Tableau 3 : Graphes testés

Le graphe est généré par un petit programme générateur de graphes appelé graph_generator. Ce programme demande à l'utilisateur le nombre de sommets ainsi que la densité et génère ensuite un fichier de graphe correspondant. La première ligne du fichier généré contient les deux valeurs n et m : le nombre de sommets et le nombre d'arêtes. Chaque ligne suivante contient une arête qui est décrite avec 3 valeurs : sommet1, sommet2 et poids. Quand le fichier du graphe est lu, le programme utilise les deux premières valeurs pour allouer la mémoire aux structures de données. Chaque arête lue devient ensuite un objet qui est placé dans une liste d'adjacence ou alors une matrice d'adjacence, dépendamment de l'algorithme en question. Ci-dessous les différents graphes utilisés et leurs nombres d'arêtes.

3.3 *Mesures du temps*

Le temps d'exécution a été mesuré à l'aide de la librairie CPerfTimer de Dean Wyant. Cette petite librairie facilite l'utilisation de l'API QueryPerformanceCounter() dans Windows en encapsulant la complexité temporelle dans un objet PerfTimer. Pour commencer les tests, il suffit d'appeler la fonction timer.Start() et pour avoir le nombre de millisecondes, il suffit de faire timer.Elapsedms(). Tous les programmes ont été mesurés sans le temps de préparation des structures de données. Quand le graphe avec 15000 sommets est testé, 180 millions objets de type arête sont créés et chacun d'entre eux apparait deux fois, car ils doivent être dans la liste d'adjacence de deux sommets. Le temps de préparation des structures de données n'est jamais mesuré. Les mesures commencent au début de l'algorithme et se terminent à la fin.

3.4 *Note sur l'algorithme de Borůvka et de Kruskal*

L'algorithme de Borůvka a été implémenté de deux façons différentes. La première implémentation utilise une gestion des ensembles disjoints ordinaire avec une simple boucle et l'autre utilise la meilleure approche : la compression de chemin avec l'union par rang.

L'algorithme de Kruskal a été implémenté en utilisant dans les deux cas le tri a comparaison Quicksort (pour plus de détails, voir Annexe II) dont la complexité temporelle dans le cas moyen est en $O(n \log n)$. Nous avons programmé plusieurs implémentations pour montrer la différence de l'ajout d'une condition d'arrêt supplémentaire à l'algorithme de Kruskal. C'est une condition qui nécessite de garder

en mémoire le nombre d'ensembles disjoints différents ainsi que le nombre de sommets visités. Quand tous les sommets sont visités et que tout est dans le même ensemble, l'algorithme peut arrêter.

3.5 *Résultats empiriques*

Voici les résultats des graphes allant de 1000 à 15000 sommets. Les valeurs représentent le temps en millisecondes (ms) du temps d'exécution. À partir de dix résultats de tests sur des graphes différents, la moyenne du temps d'exécution est calculée. Ensuite, ces moyennes sont utilisées pour visualiser la croissance de ces douze algorithmes avec un graphique. Une légende est fournie pour connaitre les détails d'implémentation de chaque algorithme. Le graphique contient directement les noms des algorithmes importants dans sa propre légende afin de faciliter la lecture.

Pour voir les résultats complets, consulter Annexe III. Un site internet a été créé pour avoir accès aux sources et aux résultats. Vu le nombre considérable de résultats obtenus, nous avons préféré de ne pas les mettre ici pour ne pas alourdir la lecture de ce mémoire.

3.5.1 Graphes creux

Voici les résultats que nous avons obtenus pour les graphes creux dont la valeur p = 0,2. Chaque taille de graphe est testée dix fois et la moyenne des temps est affichée dans ce tableau. Notons que le temps est mesuré en millisecondes (ms).

Algorithmes	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000	13000	14000	15000
B1	2	11	24	45	71	114	149	198	251	312	399	433	527	629	738
B2	4	17	40	74	117	183	243	324	422	521	670	728	885	1057	1235
B3	4	17	38	72	114	178	235	315	409	503	646	702	857	1026	1200
B4	4	19	44	84	131	205	271	362	474	582	748	814	995	1190	1394
P1	7	30	69	124	193	293	404	515	668	860	1033	1145	1340	1554	1795
P2	4	24	58	107	173	267	373	474	612	781	956	1048	1227	1416	1638
P3	3	16	39	71	115	180	250	320	415	532	643	706	823	957	1108
P4	3	16	38	70	112	175	239	303	391	503	608	659	779	903	1039
K1	13	57	135	248	396	590	817	1076	1383	1725	2102	2448	2875	3364	3885
K2	13	57	133	244	391	583	809	1061	1363	1706	2072	2425	2832	3313	3841
K3	14	59	138	254	404	599	833	1098	1404	1763	2151	2502	2939	3428	3977
K4	13	55	130	240	383	573	794	1043	1337	1673	2056	2372	2787	3261	3780

Tableau 4 : Moyennes pour les graphes creux

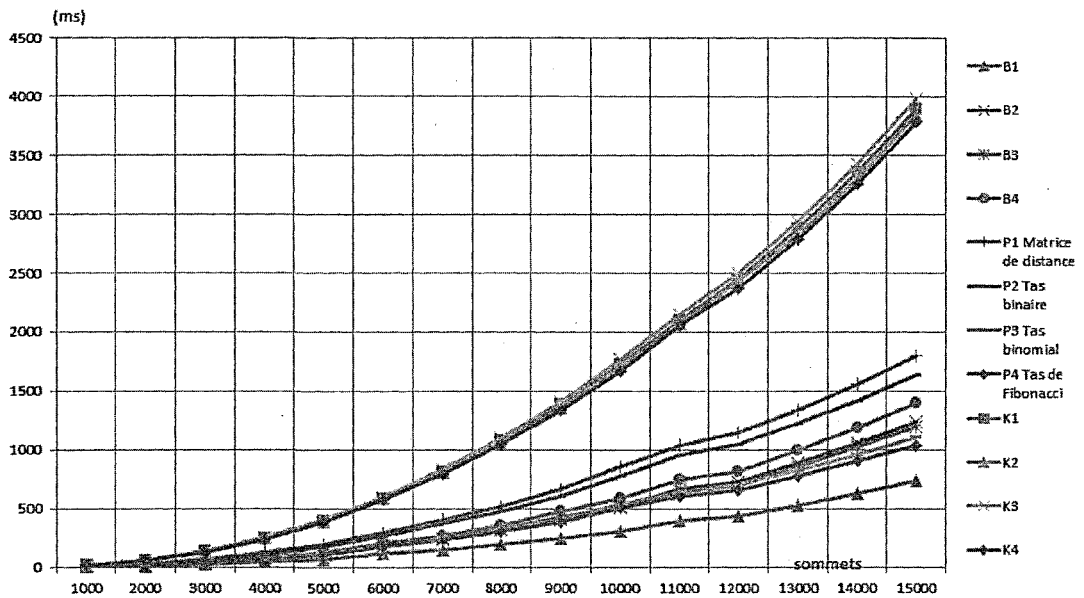


Figure 41 : Graphique des graphes creux

B1 - Algorithme de Borůvka avec gestion simple des ensembles disjoints

B2 - Algorithme de Borůvka avec compression de chemin récursive

B3 - Algorithme de Borůvka avec compression de chemin et union par rang, récursive

B4 - Algorithme de Borůvka avec compression de chemin et union par rang, itérative.

P1 - Algorithme de Prim avec matrice d'adjacence

P2 - Algorithme de Prim avec tas binaire

P3 - Algorithme de Prim avec tas binomial

P4 - Algorithme de Prim avec tas de Fibonacci

K1 - Algorithme de Kruskal avec gestion simple des ensembles disjoints

K2 - Algorithme de Kruskal avec gestion simple et condition d'arrêt

K3 - Algorithme de Kruskal avec compression de chemin

K4 - Algorithme de Kruskal avec compression de chemin et condition d'arrêt.

Ces résultats montrent que l'algorithme de Borůvka avec UNION-FIND simple et l'algorithme de Prim avec le tas de Fibonacci sont les plus rapides. Les autres versions de l'algorithme de Borůvka sont considérablement plus lentes. Nous attribuons ce problème aux facteurs constants additionnels dus à l'utilisation de l'Union-Find. Nous remarquons aussi que l'addition d'une fusion par rang (B3 et B4) n'accélère pas vraiment les performances. Nous constatons également que l'implémentation de l'UNION-FIND qui utilise la récursivité est plus rapide que notre implémentation itérative. Il semblerait que dans ce cas précis, utiliser la pile d'appel de la machine est plus rapide qu'une pile programmée.

Passons à l'algorithme de Prim. La matrice d'adjacence se montre être la pire implémentation car il faut parcourir une ligne entière de la matrice pour trouver la meilleure arête. Dans un graphe creux, les sommets n'ont pas beaucoup d'arêtes, ce qui rend cette stratégie moins productive. Le tas binaire ne se montre pas convaincant

au niveau des performances, malgré une implémentation par tableau. Le tas binomial montre de très bonnes performances, mais le tas de Fibonacci est toujours supérieur de peu. Cette différence est probablement l'œuvre de l'opération decreaseKey qui fonctionne en temps amorti constant. Si cette opération accélère réellement le tas de Fibonacci par rapport au tas binomial, alors nous verrons de plus grandes différences dans les prochains tests, lorsque la densité des graphes augmentera.

Les résultats de l'algorithme de Kruskal montrent que l'algorithme n'est pas compétitif avec Borůvka ou Prim. Entre le meilleur algorithme de Borůvka et l'algorithme de Kruskal, la différence est aussi large qu'un facteur de 4. En ce qui concerne les résultats de l'algorithme de Kruskal, le temps affiché est le temps total pour trouver le MST. Il y a donc le temps du tri additionné à l'algorithme de Kruskal. Puisque le tri prend la majorité du temps, nous observons très mal les différences entre les algorithmes d'Union-Find qui accélèrent l'algorithme de Kruskal. Pour régler ce problème, une section entière est dédiée à l'étude des algorithmes d'UNION-FIND. Ces résultats confirment les complexités temporelles respectives de ces algorithmes en $O(n \log n)$ temps. Le taux de croissance des courbes observées sur le graphique de la Figure 41 est en log-linéaire.

3.5.2 Graphes moyennement denses

Voici les résultats que nous avons obtenus pour les graphes moyennement denses dont la valeur p = 0,5. Chaque taille de graphe èst testée dix fois et la moyenne des temps est affichée dans ce tableau. Le temps est mesuré en millisecondes (ms).

Algorithmes	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000	13000	14000	15000
B1	6	25	56	96	166	244	337	436	554	674	867	1032	1216	1468	1641
B2	9	41	93	162	279	415	570	748	956	1174	1479	1784	2126	2517	2849
B3	9	40	91	158	273	404	556	730	932	1141	1442	1738	2066	2447	2768
B4	10	47	105	184	317	472	649	852	1091	1341	1694	2043	2431	2903	3267
P1	9	38	86	153	252	351	477	624	800	990	1200	1433	1714	1975	2289
P2	12	59	144	263	434	602	826	1109	1413	1764	2176	2642	3196	3863	4549
P3	8	39	95	175	291	404	555	748	952	1188	1481	1807	2171	2623	3119
P4	8	38	89	160	262	364	498	663	847	1054	1343	1638	1990	2387	2863
K1	32	137	325	594	995	1399	1939	2568	3286	4110	5010	6023	7128	8312	9648
K2	31	135	320	588	984	1382	1916	2534	3239	4064	4951	5954	7026	8216	9524
K3	33	142	335	615	1027	1444	1998	2646	3383	4230	5163	6225	7346	8582	9990
K4	31	134	317	584	975	1370	1907	2522	3218	4031	4920	5912	6981	8156	9462

Tableau 5 : Moyennes pour les graphes moyennement denses

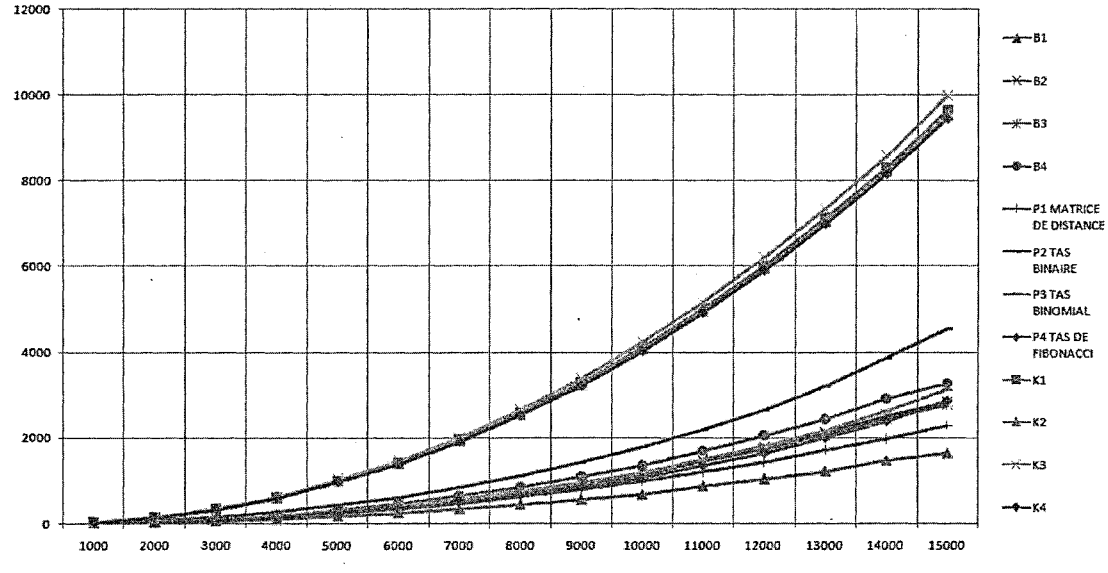


Figure 42 : Graphique des résultats pour p=0,5

Les résultats des graphes moyennement denses viennent avec de nouvelles observations. L'écart se creuse davantage avec les algorithmes de Kruskal, qui sont désormais six fois plus lents que le meilleur algorithme B1. Le tas binaire est le pire algorithme de Prim, avec des performances sévèrement en dessous du tas binomial. Le tas binomial et le tas de Fibonacci gardent leurs performances respectives. Le nouveau champion pour l'algorithme de Prim devient l'implémentation qui utilise la matrice de distances. Cette structure de données se montre être capable de supporter une forte densité beaucoup mieux que les listes d'arêtes utilisées par les trois autres algorithmes. En ce qui concerne l'algorithme de Borůvka, le meilleur algorithme de tous est encore l'algorithme qui utilise un UNION-FIND simple. Les autres algorithmes qui utilisent un UNION-FIND plus avancé souffrent des grands temps constants qui sont répétés sur un nombre log-linéaire d'arêtes. Les algorithmes B2, B3 et B4 ont donc des performances au mieux semblables à celles des algorithmes de Prim.

3.5.3 **Graphes denses**

Voici les résultats que nous avons obtenus pour les graphes denses dont la valeur p est égale à 0,8. Chaque taille de graphe est testée dix fois et la moyenne des temps est affichée dans ce tableau. Notons que le temps est mesuré en millisecondes (ms).

Algorithmes	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000	13000	14000	15000
B1	9	36	84	153	248	347	486	664	843	1035	1226	1497	1853	2058	2440
B2	14	59	138	257	420	601	842	1158	1470	1788	2177	2657	3287	3643	4299
B3	14	57	135	250	414	588	822	1130	1433	1745	2118	2589	3201	3550	4191
B4	16	66	156	290	479	683	961	1322	1679	2047	2492	3046	3777	4189	4942
P1	10	42	96	172	271	396	541	704	906	1117	1367	1626	1975	2289	2664
P2	21	96	225	411	657	959	1333	1754	2252	2889	3676	4538	5517	6710	7856
P3	14	63	150	274	443	648	900	1196	1531	1954	2498	3119	3814	4613	5535
P4	14	60	138	246	395	575	791	1042	1346	1708	2222	2794	3474	4185	4958
K1	50	218	517	945	1514	2232	3090	4077	5228	6561	8038	9598	11390	13341	15445
K2	49	215	508	933	1497	2202	3049	4031	5163	6465	7925	9478	11239	13131	15225
K3	52	228	533	979	1570	2308	3196	4226	5409	6796	8349	9989	11802	13816	15972
K4	49	214	506	930	1493	2195	3039	4015	5140	6438	7896	9436	11174	13091	15177

Tableau 6: Moyennes pour les graphes denses

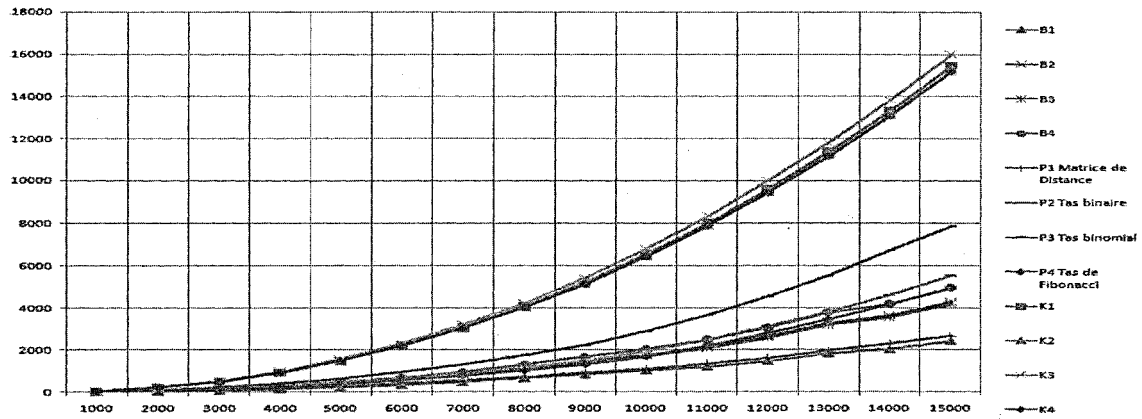


Figure 43 : Graphique des résultats pour p=0,8

Les graphes denses nous montrent les résultats d'une tendance qui avait commencée lors des graphes moyennement denses. L'algorithme de Prim qui utilise la matrice d'adjacence fait un très bon travail et se dispute la première place avec l'algorithme B1, Borůvka avec UNION-FIND Simple. La grande quantité d'arêtes présente dans les graphes denses introduit des facteurs constants qui ralentissent considérablement les bons algorithmes d'UNION-FIND utilisés par les algorithmes de Borůvka B2, B3 et B4. Les algorithmes de Prim avec tas binomial et tas de Fibonacci sont compétitifs, mais leur structure d'arêtes à pointeurs les ralentit considérablement. Ils sont plus performants avec des graphes moins denses.

Parlons maintenant de ce qui rend la matrice de distances idéale pour les graphes denses. Ces excellents résultats peuvent être attribués à plusieurs facteurs. La matrice de distances est faite pour être utilisée pour représenter de façon compacte toutes les arêtes possibles d'un sommet. Il suffit d'explorer une ligne entière de la matrice pour avoir toutes les arêtes du sommet. Le deuxième facteur est celui de la localisation des données. Une matrice de distances est plus centralisée que des vecteurs d'arêtes, ce qui lui permet d'être reproduite dans la mémoire cache du processeur et d'accélérer grandement ses performances. Les processeurs modernes ont accès à une mémoire cache qui est largement plus rapide qu'un accès en mémoire vive. Lorsque le processeur accède à une case mémoire, ce dernier va charger à l'avance les cases suivantes dans sa mémoire cache. Ainsi, les accès successifs aux cases voisines seront beaucoup plus rapides car celles-ci seront disponibles dans la cache. Lorsque nous transposons ce mode de fonctionnement à notre situation, nous nous retrouvons avec la ligne entière de la matrice dans la cache, ce qui accélère grandement les itérations de l'algorithme.

Une observation majeure concerne l'algorithme de Borůvka. Nous avons remarqué que l'algorithme de Borůvka était plus rapide sur certains tests à 15000 sommets que certains tests à 14000 sommets. Normalement, le temps d'exécution d'un algorithme augmente avec la taille du problème mais l'algorithme de Borůvka semble défier cette affirmation. Tout d'abord, voici les résultats de l'algorithme B1 pour les graphes à 14000 sommets et les graphes à 15000 sommets.

Sommets											Moyenne
14000	2203	1670	2072	2347	1744	2341	2165	1658	2218	2158	2058
15000	2692	2542	2545	1893	2480	2424	2651	1906	2606	2662	2440

Tableau 7 : Résultats intéressants de l'algorithme de Borůvka

En rouge nous avons souligné les tests qui battent le temps de certains tests à 14000 sommets (pour les tests complets, voir Annexe III). En général, nous pouvons observer que le temps des tests varie beaucoup. Pour les graphes à 15000 sommets, il y a un écart de 799 entre la plus grosse et la plus petite valeur, ce qui représente environ 32% de la valeur moyenne. Cette variance dans les résultats explique que certains tests sur 15000 sommets soient plus rapides que des tests sur 14000 sommets. Se pose donc la question, d'où vient cette variance ?

Cette variance vient du mode de fonctionnement des étapes de réduction de Borůvka. Lors d'une étape de réduction de Borůvka, chaque sommet choisit son arête incidente la plus proche, ce qui a pour effet de diminuer le nombre de sommets par un facteur de deux dans le pire des cas. Cependant, il arrive également que ce facteur soit plus favorable à cause de la structure du graphe. Puisque chaque densité de graphe est testée avec dix graphes différents, certains graphes sont accidentellement plus faciles à résoudre pour l'algorithme de Borůvka. La Figure 44 illustre ce genre de situation.

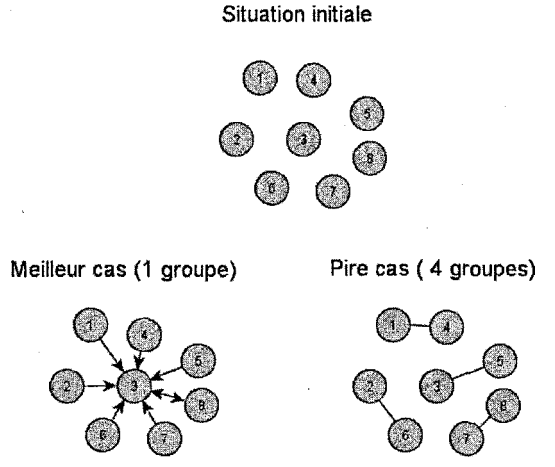


Figure 44 : Réduction de Borůvka

Cet exemple montre comment une réduction de Borůvka peut être efficace dans son meilleur cas. C'est en partie grâce à ces cas avantageux que l'algorithme de Borůvka obtient une meilleure moyenne et de meilleures performances. Les algorithmes de Prim et Kruskal n'ont pas un fonctionnement qui leur permet de prendre avantage de ces cas avantageux.

3.6 *Consommation mémoire*

Tout d'abord, il faut dire que nous avons dû apporter un changement pour que certains tests puissent fonctionner. La limite de mémoire vive sur Windows 7 pour un processus 32 bits est de 2 Go. Les trois algorithmes de Prim qui utilisent une file de priorité (tas binaire, tas binomial, tas de Fibonacci) prennent une quantité de mémoire considérable, car chaque sommet doit avoir sa propre liste d'adjacence ; cela a pour effet de doubler le nombre d'arêtes en mémoire. Ces programmes fermaient

donc tout seuls lorsqu'ils dépassaient la limite de 2 Go. Les options qui s'offraient à nous étaient soit de recompiler chaque programme en 64 bits ou alors recompiler en 32 bits avec l'option /LARGEADDRESSAWARE. Cette option permet à un processus 32 bits d'utiliser 4 Go de mémoire sur Windows 64 bits. Après avoir utilisé cette option, nos trois programmes ont utilisé au maximum 3.4 Go de mémoire vive. Ci-dessous, les mesures de consommation mémoire des différents programmes pour le plus gros test : 15 000 sommets avec densité 0.8 et 101 246 635 arêtes.

Algorithmes	N=15000
B1	1.58
B2	1.58
B3	1.58
B4	1.58
P1	0.95
P2	3.4
P3	3.4
P4	3.4
K1	1.6
K2	1.6
K3	1.6
K4	1.6

Tableau 8 : Consommation mémoire en Go

3.7 *Algorithmes de Union-Find pour Kruskal*

Dans cette section nous allons étudier les algorithmes d'Union-Find à travers l'algorithme de Kruskal. Dans la section précédente, nous avons comparé Kruskal aux autres algorithmes en considérant le temps total de son exécution, c'est-à-dire le temps du tri des arêtes additionné au temps de l'algorithme de Kruskal. Le temps nécessaire pour trier les arêtes est tellement dominant que nous ne pouvons pas voir

de différence entre les quatre versions de l'algorithme et leurs optimisations respectives. Dans cette section nous avons remesuré les quatre algorithmes de Kruskal en employant la même méthodologie que précédemment. Nous n'avons donc tenu compte que du temps d'exécution de l'algorithme UNION-FIND à l'intérieur de Kruskal.

Rappelons rapidement son fonctionnement. Essentiellement, nous avons un vecteur d'arêtes trié en ordre croissant. L'algorithme va passer une fois sur chaque arête et va questionner son algorithme UNION-FIND à chaque fois pour savoir s'il faut ajouter cette arête ou l'ignorer. La structure d'UNION-FIND gère les ensembles disjoints. À chaque arête visitée, cette structure peut créer un nouveau groupe, fusionner deux groupes ou répondre que l'arête doit être ignorée, car elle crée un cycle. Dans notre expérimentation, nous avons testé les algorithmes suivants.

1. K1. Algorithme de Kruskal avec Union-Find simple.
2. K2. Algorithme de Kruskal avec Union-Find simple avec break
3. K3. Algorithme de Kruskal avec Compression de Chemin
4. K4. Algorithme de Kruskal avec Compression de Chemin et break.

L'algorithme K1 utilise un algorithme d'Union-Find simple. Un vecteur de taille n (le nombre de sommets) est géré et chaque case de ce vecteur représente le groupe dans lequel se trouve ce sommet. Lorsqu'il faut faire une fusion entre deux groupes, nous choisissons arbitrairement le groupe parent et nous passons sur ce vecteur en changeant toutes les valeurs du groupe enfant pour le groupe parent. Cette boucle nécessite un temps en O(n).

L'algorithme K2 utilise le même algorithme que K1 mais ce dernier s'arrête quand le MST est généré. La condition d'arrêt vérifie que tous les sommets ont été

visités et qu'il n'y a qu'un seul groupe. Cette condition utilise deux variables. Une variable qui compte le nombre de sommets restants et une variable qui représente le nombre de groupes. Vérifier cette condition prend un temps constant.

L'algorithme K3 utilise l'algorithme UNION-FIND avec union par rang et compression de chemin, comme détaillé dans la revue de la littérature. L'algorithme K4 représente l'algorithme K3 avec la condition d'arrêt.

3.7.1 UNION-FIND sur les graphes creux

Voici les résultats que nous avons obtenus pour les graphes creux dont la valeur p=0,2. Chaque taille de graphe est testée cinq fois et nous affichons la moyenne des temps dans ce tableau. Le temps est mesuré en millisecondes (ms).

Algorithm	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000	13000	14000	15000
K1	0,48	1,95	4,35	7,78	12,30	17,63	25,01	32,05	40,19	50,74	61,59	73,42	86,99	100,65	117,62
K2	0,29	1,14	2,50	4,50	7,18	10,30	14,48	18,76	23,18	29,52	35,20	41,68	49,30	56,94	66,13
K3	0,87	3,40	7,74	13,61	21,00	30,06	42,20	53,92	68,36	85,76	104,22	126,93	148,11	173,59	200,48
K4	0,07	0,15	0,22	0,30	0,39	0,52	0,57	0,68	0,80	0,90	1,06	1,01	1,18	1,35	1,40

Tableau 9 : UNION-FIND pour les graphes creux

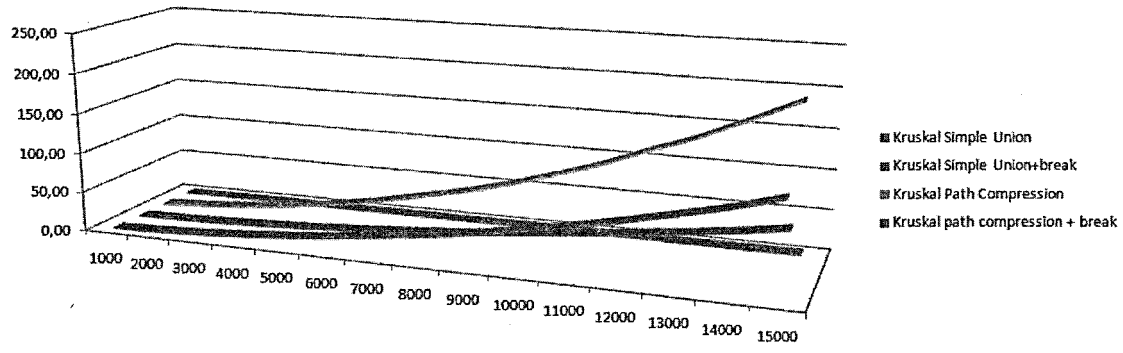


Figure 45 : UNION-FIND pour les graphes creux

3.7.2 UNION-FIND sur les graphes moyennement denses

Voici les résultats que nous avons obtenus pour les graphes moyennement denses dont la valeur p = 0,5. Chaque taille de graphe est aussi testée cinq fois et la moyenne des temps est affichée dans ce tableau. Notons que le temps est mesuré en millisecondes (ms).

Algorithm	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000	13000	14000	15000
K1	0,80	3,22	7,18	12,72	19,41	28,10	38,30	49,79	64,11	80,09	96,96	115,91	138,37	160,75	190,33
K2	0,30	1,13	2,55	4,57	7,02	9,84	13,67	17,74	22,34	27,26	33,83	39,62	46,69	54,94	63,34
K3	2,20	8,60	19,48	33,85	50,99	72,92	98,17	128,69	165,91	204,67	251,90	315,60	370,11	446,58	553,60
K4	0,07	0,16	0,23	0,29	0,37	0,48	0,57	0,66	0,72	0,84	0,90	0,99	1,07	1,18	1,45

Tableau 10 : UNION-FIND pour les graphes moyennement denses

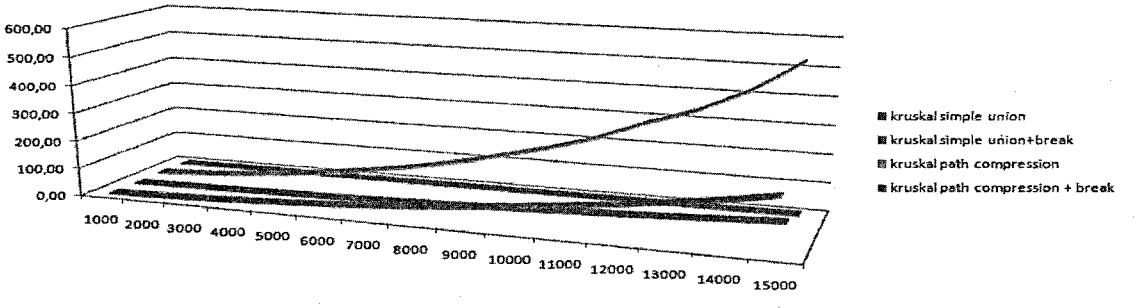


Figure 46 : UNION-FIND pour les graphes moyennement denses

3.7.3 UNION-FIND sur les graphes denses

Voici les résultats que nous avons obtenus pour les graphes denses de valeur p=0,8. Chaque taille de graphe est également testée cinq fois et la moyenne des temps est affichée dans ce tableau. Notons que le temps est mesuré en millisecondes (ms).

Algorithm	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000	13000	14000	15000
K1	1,10	4,40	9,79	17,63	27,88	39,83	55,68	73,25	94,94	118,44	150,70	179,02	211,63	252,36	351,05
K2	0,29	1,11	2,50	4,54	7,18	10,17	14,33	18,54	23,14	28,97	34,54	41,80	48,66	56,28	65,47
K3	3,50	13,39	30,39	53,31	83,31	119,21	164,00	226,69	316,25	398,57	445,44	569,59	652,09	797,09	873,44
K4	0,07	0,15	0,23	0,33	0,40	0,47	0,57	0,64	0,79	0,84	0,91	1,01	1,12	1,14	1,49

Tableau 11 : UNION-FIND pour les graphes denses

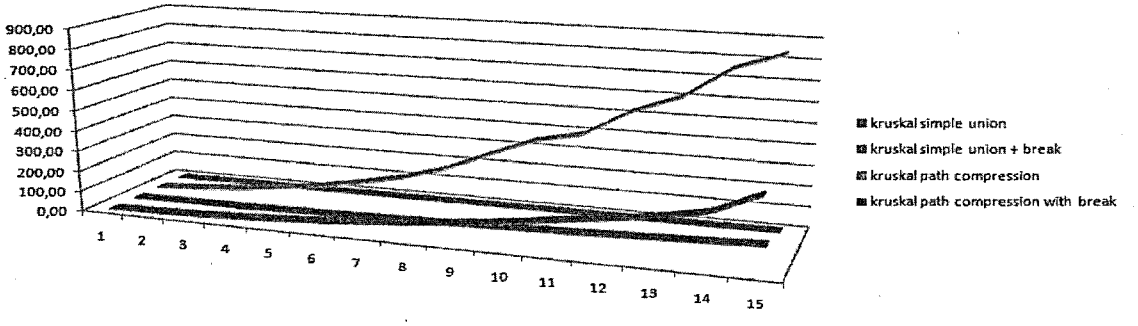


Figure 47: UNION-FIND pour les graphes denses

3.8 *Discussion sur les résultats*

Notons d'emblée que les résultats produits par les algorithmes d'UNION-FIND sont très intéressants. Contrairement à la section précédente, il n'est pas nécessaire ici de discuter des résultats pour chaque densité de graphe, car celle-ci n'a pas influencé le classement de ces algorithmes. Tout d'abord, nous voulons souligner la très bonne performance de l'algorithme UNION-FIND qui utilise la compression par chemin, l'union par rang et la condition d'arrêt (Algorithme K4). C'est l'algorithme d'UNION-FIND le plus rapide dans la littérature, avec une complexité amortie égale à la fonction Ackermann inverse $O(\alpha(m))$. C'est une fonction qui croit très lentement et nous pouvons le voir dans les graphiques de résultats. Il a fallu présenter les résultats sous

forme d'un graphique 3D pour pouvoir voir la droite correspondant à la croissance de cet algorithme, car en deux dimensions elle n'apparaissait même pas. En revanche, l'algorithme K3 est l'algorithme le moins rapide. Pourquoi donc ?

En fait, l'UNION-FIND avec compression de chemin est un algorithme efficace, mais le facteur constant nécessaire à son fonctionnement additionné à des millions d'arêtes devient un temps considérable en pratique. Nous avons donc le même problème qui faisait en sorte que l'UNION-FIND avec compression de chemin était moins performant pour l'algorithme de Borůvka. La version K4 règle ce problème, car elle traite considérablement moins d'arêtes avec la présence de la condition d'arrêt. Cette condition d'arrêt permet à la vraie performance de l'algorithme K4 d'être révélée. Nous observons également que l'ajout d'une condition d'arrêt accélère grandement chaque algorithme de Kruskal. Chaque graphe testé est généré par ordinateur et même les graphes à faible densité sont bien connectés. Nous ne sommes donc jamais dans le pire des cas où l'arête la plus lourde est nécessaire au MST. C'est une bonne optimisation. L'algorithme K2 (UNION-FIND simple avec condition d'arrêt) offre également des performances intéressantes, ne prenant que 65 ms pour un graphe très dense avec 15000 sommets. Cependant, l'algorithme K4 ne prend que 1,5 ms, ce qui fait de ce dernier l'incontestable meilleur choix.

CONCLUSION

La problématique sur laquelle nous nous sommes penchés dans ce mémoire est celle de l'arbre couvrant minimal. Cette problématique est très connue en théorie des graphes. Les algorithmes principaux que nous avons utilisés pour sa résolution sont ceux de Kruskal, Borůvka et Prim. Les algorithmes d'UNION-FIND et les files de priorité sont également présentés, car ces derniers sont utilisés par les algorithmes que nous avons utilisés. Les algorithmes de Kruskal, Borůvka, Prim et UNION-FIND ont été testés extensivement dans ce travail. L'algorithme de Prim a été testé avec une matrice de distances, un tas binaire, un tas binomial et un tas de Fibonacci. L'algorithme de Kruskal a été testé avec plusieurs optimisations d'UNION-FIND et une condition d'arrêt. L'algorithme de Borůvka a été testé avec plusieurs optimisations d'UNION-FIND. Nous avons testé tous ces algorithmes avec une batterie de tests qui teste les limites d'un processeur moderne, certains tests prenant plus d'une heure à compléter.

Nos conclusions sont multiples. L'algorithme de Kruskal est l'algorithme le plus simple, mais il est le moins performant à cause du tri préalable des arêtes. Nous avons vu que l'ajout d'un bon algorithme UNION-FIND et d'une condition d'arrêt accélèrent d'une manière significative sa vitesse d'exécution. Ce sont de bonnes optimisations et l'algorithme de Kruskal est extrêmement rapide, le plus rapide de tous, si les arêtes

sont déjà triées. Cependant, si les arêtes ne sont pas triées, il faut les trier avec un algorithme de tri standard, ce qui va générer des performances médiocres.

L'algorithme de Prim est un algorithme qui opère différemment. Au lieu de construire plusieurs arbres couvrants en parallèle pour ensuite les fusionner comme font Kruskal et Borůvka, l'algorithme de Prim construit l'arbre couvrant minimal en faisant grossir un seul arbre. Ce qui enlève la difficulté de gestion des ensembles disjoints et qui fait que cet algorithme à une meilleure complexité temporelle. Couplé avec le tas de Fibonacci, l'algorithme de Prim atteint une complexité temporelle en $O(n\log n+m)$. Dans nos tests, l'algorithme de Prim est très performant. Le tas de Fibonacci se montre excellent pour les graphes peu denses. En revanche, ses performances diminuent lorsque les graphes sont plus denses. Nous avons en effet constaté que les listes d'adjacences sont moins intéressantes en pratique. Plus la densité du graphe augmente, plus ces algorithmes souffrent de problèmes de performance. La cause de ce ralentissement est dans l'architecture de la machine. La machine est meilleure pour itérer sur des tableaux plutôt que des structures à pointeurs. Les résultats de l'algorithme de Prim avec la matrice de distance viennent confirmer cette affirmation, car cet algorithme qui fonctionne avec une matrice a une performance excellente sur les graphes denses.

L'algorithme de Borůvka s'est montré être l'algorithme le plus performant. Cet algorithme se démarque des autres par ses performances excellentes sur les trois types de graphes. L'algorithme de Borůvka, tout comme Kruskal, construit plusieurs

arbres couvrants en parallèle et doit gérer des ensembles disjoints. Pour pouvoir observer l'avantage qui vient avec l'utilisation d'un meilleur algorithme d' UNION-FIND, nous avons testé Borůvka avec plusieurs algorithmes de UNION-FIND. Nous avons été surpris de constater que l'utilisation du meilleur algorithme d' UNION-FIND, celui avec la compression de chemin, donnait de moins bonnes performances que l'algorithme d' UNION-FIND simple. Nous avons compris que la disproportion entre requêtes et fusions qui existe dans l'algorithme de Borůvka n'était pas idéale car les requêtes sont plus lentes avec l'UNION-FIND par compression de chemin. Les fusions étant beaucoup plus rares que les sélections dans l'algorithme de Borůvka, l'algorithme d'UNION-FIND simple s'est montré plus performant pour Borůvka.

Nous nous sommes ensuite intéressés davantage aux algorithmes UNION-FIND en analysant la version contenue dans l'algorithme de Kruskal séparément. Nos résultats confirment la remarquable efficacité de l'UNION-FIND avec compression de chemin. Les graphiques générés montrent une croissance quasi-nulle du temps d'exécution entre les graphes a 1000 sommet et les graphes à 15000 sommets, ce qui correspond à la complexité algorithmique d'une fonction Ackermann inverse.

Avec tous ces résultats, nous sommes confiants que ce travail propose une étude expérimentale concise des algorithmes de Kruskal, Prim, Borůvka et UNION-FIND, ainsi que des différentes structures de données utilisées par ces derniers dont le tas de Fibonacci, tas binomial, tas binaire et matrice de distance. Nous pensons que ce

travail propose une bonne introduction au problème de l'arbre couvrant minimal, ainsi que des résultats pertinents à l'utilisation de ces algorithmes en pratique.

ANNEXE I

COMPLEXITÉ ALGORITHMIQUE

Comme l'évaluation de la qualité des algorithmes a été le fil conducteur tout au long de ce mémoire, nous avons pensé qu'il serait judicieux d'inclure quelques concepts de complexité algorithmique.

La complexité temporelle réfère à l'évaluation de l'efficacité d'un algorithme. Cette efficacité passe par l'évaluation du temps mis pour transformer les données du problème considéré en un ensemble de résultats.

Nous ne sommes pas intéressés par une performance précise mesurée en unité de temps comme les millisecondes. Nous cherchons plutôt à exprimer la performance de l'algorithme comme une fonction mathématique dans la variable est la taille des données. Nous savons qu'une fonction exponentielle croit beaucoup plus rapidement qu'une fonction linéaire. Certains algorithmes vont faire le même travail, mais avec des complexités temporelles très différentes.

L'analyse de la complexité algorithmique s'intéresse à deux facettes d'un algorithme : le temps et l'espace. Le temps représente le temps d'exécution et l'espace représente l'espace mémoire utilisé pour transformer les données d'entrée en résultats. Les termes complexité temporelle et complexité spatiale sont également utilisées. Ces deux facettes sont très importantes. Néanmoins, dans le travail que nous

avons présenté sur l'arbre couvrant minimal, nous nous sommes intéressés beaucoup plus à la complexité temporelle. Lorsque nous étudions la performance d'un algorithme, souvent nous décrivons son comportement suivant plusieurs cas d'exécution possibles : le pire cas, le meilleur cas et le cas moyen. Tout en expliquant ces concepts, nous allons introduire des exemples d'algorithmes avec la notation O. Cette notation permet de décrire la performance d'un algorithme en étudiant sa croissance par rapport aux données d'entrée. Si un algorithme de recherche doit faire autant de comparaisons que de données d'entrée, nous disons que ce dernier à une croissance linéaire. Cette croissance est décrite comme une fonction s'exprimant à l'aide de la taille des données d'entrée. La complexité algorithmique est souvent exprimée à l'aide des symboles (O, Ω, Θ). Le symbole O se lit « ne croit pas plus vite que », le symbole Ω se lit « croit au moins aussi vite que » et le symbole Θ se lit « croit aussi vite que ». Généralement, le symbole O est le plus utilisé dans les analyses de complexité algorithmique. La notation $O(n^2)$ se lit de la façon suivante : l'algorithme ne croit pas plus vite que la fonction n^2. Généralement, nous distinguons quatre sortes de complexité : pire cas, meilleur cas, cas moyen et le cas amorti.

LA COMPLEXITE DANS LE PIRE CAS

Le pire cas survient lorsque l'algorithme à pris le chemin le plus long pour produire son résultat. Généralement, quand nous discutons d'un algorithme, nous allons discuter de son exécution dans le pire cas. Le pire cas est intéressant, car il nous renseigne sur le temps maximal pris par l'algorithme pour terminer son exécution. Les systèmes à temps réel ont particulièrement besoin de performances prévisibles, car leurs opérations doivent pouvoir s'exécuter dans une tranche de temps définie.

Exemple : Recherche Linéaire

Nous avons un tableau de n entiers et nous voulons connaître la position d'un élément x dans le tableau. Pour simplifier l'analyse, nous supposons que l'élément **x** existe dans le tableau. Dans le pire des cas, l'élément est en dernière position du tableau. Il faudra donc parcourir tout le tableau pour le trouver.

```
Fonction Chercher(x, tableau, n)
{
 Pour i=0 à n-1
   SI tableau[i] == x
     return i;
 return -1; //pas trouvé
} //fin de la fonction
```

Algorithme 14 : recherche d'un élément

Lorsque nous analysons le temps d'exécution d'un algorithme, nous considérons que des instructions élémentaires comme l'addition, la soustraction, la division et la multiplication prennent un temps constant. On suppose également que

les instructions de comparaisons et de mouvement de données sont effectuées en un temps constant, noté $O(1)$.

Maintenant, pour analyser le temps d'exécution de cet algorithme, nous devons calculer combien d'opérations sont faites en pire cas. Dans le pire des cas, l'élément se trouve à la fin du tableau. Il y aura donc n opérations de comparaison, $(n-1)$ additions et 1 pour le return. Nous allons toujours simplifier la complexité temporelle en ignorant les facteurs constants, ce qui donne une complexité dans le pire cas en $O(n)$.

LA COMPLEXITE DANS LE MEILLEUR CAS

Le meilleur cas est beaucoup moins utilisé, car cette mesure ne donne pas une assurance sur la performance de l'algorithme. C'est une mesure qui permet de voir qu'un algorithme va performer beaucoup mieux sur certaines entrées. Si nous reprenons notre exemple de recherche dans un tableau, le meilleur cas survient lorsque notre entrée contient l'élément cherché dans la première case du tableau. L'algorithme fait donc un nombre constant d'opérations. La complexité temporelle dans le meilleur cas est donc O(1).

LA COMPLEXITE DANS LE CAS MOYEN

Nous avons vu que les algorithmes peuvent avoir des complexités temporelles différentes dépendamment de l'instance utilisée. Comme nous ne connaissons pas l'instance sur laquelle un algorithme sera exécuté, généralement, nous faisons des

suppositions sur la distribution des données constituant une instance. En procédant de cette manière, nous arrivons à calculer le temps moyen (représentant l'espérance mathématique) d'exécution de l'algorithme en question. La distribution la plus utilisée dans l'analyse des algorithmes est la répartition uniforme. Si nous reprenons notre exemple de recherche dans un tableau, nous pouvons calculer un cas moyen. Dans ce qui suit, avec la même probabilité, nous supposons que l'élément recherché peut être n'importe où dans le tableau. Chaque case du tableau i a donc une probabilité $P_i = \frac{1}{n}$ de contenir l'élément recherché.

La complexité temporelle moyenne est calculée en faisant la somme de toutes les complexités allant de 1 à N, chacune associée à sa probabilité d'apparition (qui est dans ce cas de 1/n). La complexité dans le cas moyen est donc :

$$\sum_{i=1}^{n} P_i \, i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2} \approx \frac{n}{2}$$

Autrement dit, nous devons donc faire en moyenne $\frac{n}{2}$ comparaisons pour trouver un élément dans le tableau. Ce qui donne en notation O une complexité temporelle en $O(n)$ dans le cas moyen.

LA COMPLEXITE DANS LE CAS AMORTI

L'analyse amortie est une technique d'analyse qui permet d'avoir une borne supérieure en pire cas pour un ensemble d'opérations qui modifient l'état de la structure de donnée. Lorsque plusieurs chemins d'exécutions existent dans un algorithme, ces chemins sont pris dépendamment de l'état de la structure de données.

Ces chemins n'ont pas forcément le même temps d'exécution. Avec l'analyse amortie, nous définissons la complexité temporelle d'un algorithme comme la moyenne arithmétique de ses différents chemins d'exécution. Dans l'analyse en cas moyen, nous nous intéressons à la distribution des entrées des données d'entrée afin de pouvoir déduire une complexité en cas moyen. L'analyse amortie ne se soucie pas des données d'entrée, mais plutôt du comportement amorti du même algorithme/opération sur plusieurs utilisations. Trois techniques sont utilisées pour entreprendre l'analyse amortie d'un algorithme donné. Pour plus de détails, consulter [Cormen, Leiserson, Rivest and Stein 2009].

1. METHODE D'AGREGATION

Prenons comme exemple le problème du tableau dynamique, un tableau capable d'augmenter sa taille automatiquement. Lorsque le tableau est plein, la prochaine insertion entraîne un comportement en pire cas qui fonctionne de la façon suivante : un nouveau tableau deux fois plus gros est créé, les éléments du tableau précédents sont déménagés dans le nouveau tableau, l'ancien tableau est détruit, et le nouvel élément est inséré à la fin du nouveau tableau.

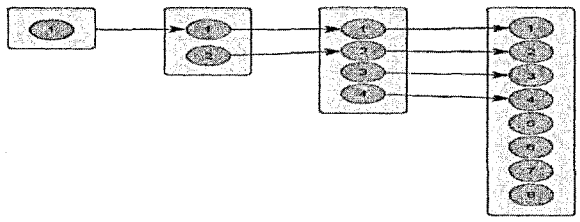


Figure 48 : Croissance du tableau dynamique

Quand nous insérons un élément dans ce tableau, il y a deux chemins d'exécutions possibles. Dans le premier cas, une place est libre pour notre élément et il est inséré en temps constant. Dans le deuxième cas, le tableau est plein. En supposant que l'obtention d'un nouveau tableau plus grand se fait en un temps constant, il faut ensuite copier les n éléments de notre ancien tableau dans le nouveau tableau et insérer notre nouvel élément, ce qui prend un temps n+1.

```
Inserer(tableau, x) {
  SI taille[tableau] = nombre[tableau]{
        AgrandirTableau(tableau) ;
        Inserer(tableau,x) ;
   }
  SINON {
        tableau[nombre] = x ;
        taille[tableau] ++ ;
  }
}
```

Algorithme 15 : tableau dynamique

Regardons maintenant la complexité temporelle en pire cas de cette structure de données. Le pire cas de l'opération insertion est en O(n) temps Si nous voulons insérer n éléments, le travail total sera donc en $O(n^2)$ temps. Cette analyse ne rend pas justice à cet algorithme, car le pire cas se produit assez rarement. Avec un nombre n d'insertions, la réallocation ne se fait que log n fois. La somme de toutes les réallocations est $\sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j$.

Cette somme est une série géométrique qui, une fois simplifiée, est égale à environ 3n. Donc pour n éléments, toutes les réallocations prennent un temps linéaire.

Pour obtenir le temps amorti avec la méthode de l'agrégat, nous divisons le temps total par le nombre d'opérations. Le temps amorti pour chaque insertion est donc en $O(n)/n = O(1)$ temps.

2. METHODE COMPTABLE

Dans la méthode d'agrégation, nous cherchons à calculer le temps total que prennent les opérations coûteuses pour un nombre n d'opérations et ensuite le temps est amorti sur n opérations. Ce qui permet pour l'exemple du tableau dynamique de trouver que le temps des réallocations prend $O(n)$ temps pour n insertions et donc que ce temps peut être amorti sur les n insertions, ce qui fait en sorte que l'opération de l'insertion nécessite $O(1)$ temps amorti.

Dans la méthode du comptable, il faut mettre un coût sur l'opération amortie par rapport au temps réel. Le temps sauvé grâce au coût additionnel est ensuite utilisé pour gérer les opérations coûteuses lorsqu'elles surviennent. En d'autres mots, l'argent économisé par les opérations amorties va dans un compte en banque et lorsqu'une opération coûteuse survient, la balance du compte est utilisée pour payer. Le but de la méthode comptable est de trouver le meilleur prix à charger afin que le compte en banque n'aille jamais dans le négatif et qu'il soit capable de tout payer. De nombreuses valeurs de coûts peuvent être utilisées pour faire fonctionner l'opération amortie et plus la valeur est petite et fonctionne, meilleure sera la complexité amortie.

Revenons à l'exemple du tableau dynamique. Supposons que nous possédons ce tableau et que nous chargeons un coût amorti $C_i = 3\$$ pour y insérer une valeur et que pour nous, le coût d'insertion est de $C_i = 1\$$ seulement. Nous faisons donc un profit de 2$ à chaque insertion. Ces profits seront utilisés plus tard pour payer le gros déménagement vers un plus grand tableau. Montrons par induction que charger 3$ est idéal pour la méthode comptable. La Figure 49 est un exemple qui montre le comportement de la méthode du comptable sur plusieurs insertions. Dans chaque case du tableau, nous enregistrons l'argent restant à cette opération. La somme des restants du tableau représente le compte en banque.

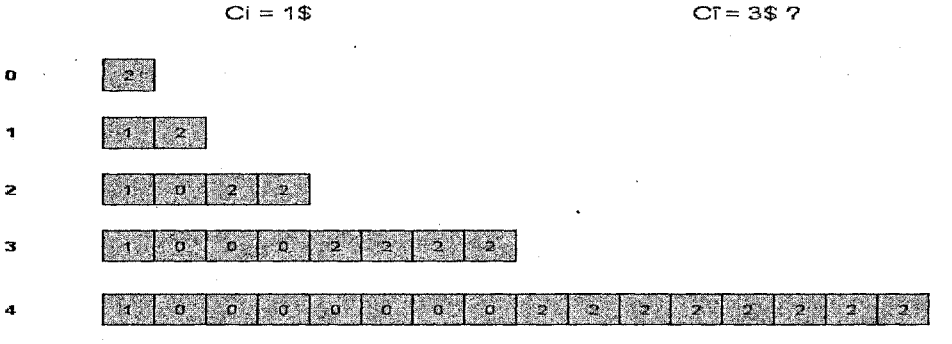


Figure 49: Méthode Comptable

À l'étape 0, nous commençons par charger 3$ pour insérer le premier élément. Il coûte 1$ à insérer dans son tableau et donc il reste 2$. Ensuite nous devons grossir le tableau pour insérer un autre élément. Le tableau a maintenant une taille deux, et le 2$ économisé est suffisant pour payer le déménagement qui coûte 1$. Il reste donc 1$ à cet élément. Nous insérons un nouvel élément pour 3$ et il reste 2$. En 2, nous devons grossir à nouveau le tableau et nous remarquons que le 2$ économisé par le

dernier élément est suffisant pour payer le déménagement des deux éléments. En 3, après avoir rempli ce tableau, nous devons encore déménager vers un tableau deux fois plus grand. Nous pouvons remarquer ici que la nouvelle moitié du tableau est toujours capable de payer pour son propre déménagement et celui de l'autre moitié. La même chose se produit à l'itération 4 et ainsi de suite pour une itération n. Le 1$ qui est transféré à tous les déménagements signifie que le premier élément n'avait pas besoin de déménager un autre élément, il avait donc gardé 1$. Nous pouvons donc faire charger 2$ à la première itération et 3$ aux suivantes.

3. METHODE DU POTENTIEL

La méthode du potentiel consiste à déduire le coût amorti d'une opération à partir des changements de potentiel entrainés par certaines opérations. C'est un fonctionnement différent de la méthode comptable dans laquelle nous assignons directement une valeur à l'opération et nous faisons les tests pour voir si cette valeur fonctionne.

Le potentiel du tableau T analysé est noté par $\Phi(T)$. La difficulté dans l'utilisation de la méthode du potentiel est de trouver la fonction qui représente l'état de la structure de donnée après l'opération. Lorsque le tableau vient d'être étendu, nous avons un potentiel de zéro, car le tableau peut recevoir de nouveaux éléments. Soit n le nombre d'éléments dans le tableau et t la taille du tableau, et considérons la fonction $\Phi(T) = 2n - t$.

Illustrons cette fonction sur l'exemple suivant :

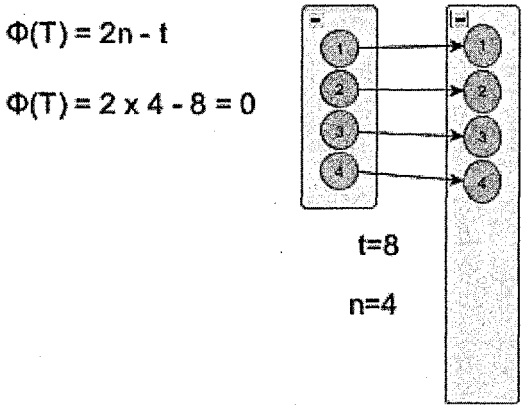


Figure 50 : Test fonction potentiel

Les autres états peuvent être déduits facilement à partir de cette première fonction. Quand la table est pleine, la taille = le nombre d'éléments, donc :

$$\Phi(T) = 2n - n = n.$$

Puisque le potentiel varie de 0 à *n*, le potentiel n'est donc jamais négatif, et nous pouvons donc noter :

$$\Phi(T) = 2n - t \geq 0.$$

Nous allons maintenant déduire le coût amorti de l'insertion à partir de la fonction de changement de potentiel.

Soit c_i le coût amorti, c le coût réel de l'opération, n le nombre d'éléments de la table, t la taille de la table et Φi le potentiel de la table après l'itération i.

Chemin d'exécution 1 : l'insertion n'entraîne pas une expansion de la table

La taille du tableau ne change pas, seul le nombre change. Par conséquent,

$$\begin{aligned} c_i &= c + \Phi_{i+1} - \Phi_i \\ &= 1 + (2(n+1) - t) - 2n - t \\ &= 1 + (2n + 2 - t) - (2n - t) \\ &= 1 + 2 = 3. \end{aligned}$$

Explication : Ici, puisqu'il n'y a pas d'expansion de la table, nous prenons les deux équations de potentiel qui représentent une table non pleine. La première équation Φ_{i+1} a un élément de plus, nous faisons donc la substitution en ajoutant 1.

Chemin d'exécution 2 : L'insertion entraîne une expansion de la table

Dans ce cas, la taille du tableau double. Ce qui signifie que $t = 2n$. Il s'en suit donc :

$$\begin{aligned} c_i &= c + \Phi_{i+1} - \Phi_i \\ &= 1 + (2(n+1) - 2n) - (2n - 2n) \\ &= 1 + (2n + 2 - 2n) - (2n - 2n) \\ &= 3. \end{aligned}$$

ANNEXE II

ALGORITHME QUICKSORT

L'algorithme Quicksort est l'algorithme que nous avons utilisé avec l'algorithme de Kruskal pour trier les arêtes. L'algorithme nécessite en moyenne $O(n\log n)$ comparaisons pour trier n éléments, ce qui est la meilleure complexité temporelle possible pour un algorithme de tri par comparaisons [Cormen, Leiserson, Rivest and Stein 2009].

D'autres algorithmes comme Mergesort et Heapsort partagent la même complexité temporelle, mais ils sont plus lents en pratique, ce qui fait de Quicksort un choix plus judicieux. L'algorithme Quicksort peut également trier les données sur place, ce qui évite la copie de données dispendieuse.

L'algorithme Quicksort repose sur le concept du pivot. Chaque itération de l'algorithme choisit un pivot et partitionne ensuite toutes les données en fonction de ce pivot. On choisit donc un élément qui sera le pivot et nous créons deux partitions. La première partition contiendra les éléments plus petits que le pivot et la deuxième partition contiendra les éléments plus grands que le pivot. On répète ensuite le processus récursivement pour chaque partition jusqu'à obtenir le cas de base. Le cas de base est un sous-problème avec deux éléments et l'algorithme retourne toujours la bonne séquence triée pour deux éléments.

Dans cette implémentation de Quicksort, nous allons présenter l'algorithme qui trie les données sur place. Cet algorithme utilise un espace auxiliaire en $O(\log n)$, car les appels récursifs utilisent un espace constant pour passer les deux variables début et fin. Le pivot est toujours choisi comme l'élément du milieu du vecteur.

```
quicksort(a[], int debut, int fin){
  i = debut;
  j = fin ;
  pivot = a[(debut+fin) /2] ;//on prend l'élément du milieu comme valeur de pivot

  TANT QUE i <= j //tant que que les deux pointeurs ne se sont pas dépassés
        //Faire avancer le pointeur i (qui commence au debut du tableau)
        tant que ( a[i] < pivot ) i++ //On cherche un élément plus grand que le pivot.
        Pire cas : Le pointeur avance jusqu'au pivot

          //Même chose de l'autre côté

            tant que ( a[j] > pivot) j-- ;
                //échange
                SI i <= j
                {
                   tmp = a[j]; a[j] = a[i];
                   a[i] = tmp; i++; j--;
                }

        //Appels récursifs
        SI debut < j
            quicksort(a, debut, j) ;

        SI i < fin
           quicksort(a, i, fin) ;

} //fin de la fonction
```

Algorithme 16 : Quicksort

Pour plus de détails sur l'analyse de la complexité temporelle de Quicksort, voir [Cormen, Leiserson, Rivest and Stein 2009].

ANNEXE III

CODE SOURCE DES PROGRAMMES ET RÉSULTATS

Le code source de tous les programmes en C++ est disponible sur mon github. Les résultats de l'expérimentation sont également disponibles dans un fichier Microsoft Excel.

Voir :

edmondlachance.com
ou
http://github.com/mitchi

Voici les programmes et scripts disponibles :

1. Algorithme de Borůvka avec gestion simple des ensembles disjoints
2. Algorithme de Borůvka avec compression de chemin récursive
3. Algorithme de Borůvka avec compression de chemin et union par rang, récursive
4. Algorithme de Borůvka avec compression de chemin et union par rang, itérative.
5. Algorithme de Prim avec matrice d'adjacence
6. Algorithme de Prim avec tas binaire
7. Algorithme de Prim avec tas binomial
8. Algorithme de Prim avec tas de Fibonacci
9. Algorithme de Kruskal avec gestion simple des ensembles disjoints
10. Algorithme de Kruskal avec gestion simple et condition d'arrêt
11. Algorithme de Kruskal avec compression de chemin

12. Algorithme de Kruskal avec compression de chemin et condition d'arrêt.
13. Générateur de graphes
14. Scripts d'exécution des tests

BIBLIOGRAPHIE

BERENBAUM, A. (1998): Experimental study of performance of minimum spanning tree algorithms. *Master thesis, Rochester Institute ofTechnology*, 124.

CORMEN, T.H., LEISERSON, C.E., RIVEST, R.L. AND STEIN, C. (2009): *Introduction to Algorithms, Third Edition*. The MIT Press.

FREDMAN, M.L. AND TARJAN, R.E. (1987): Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34*, 596-615.

GRAHAM, R.L. AND HELL, P. (1985): On the history of the minimum spanning tree problem. *Annals of the History of Computing 7*, 43-57.

KARGER, D., KLEIN, P. AND TARJAN, R. (1995): A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. *Journal of the ACM 42*, 321-328.

KRUSKAL, J. (1956): On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society 7*, 48-50.

PRIM, R.C. (1957): Shortest connection networks and some generalizations. *Bell System Technology Journal 36*, 1389-1401.

TARJAN, R.E. (1975): Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM 22*, 215-225.

VUILLEMIN, J. (1978): A data structure for manipulating priority queues. *Commun. ACM 21*, 309-315.